21世纪高等学校计算机类课程创新规划教材 · 微课版



Python程序设计与算法基础教程（第2版）

微课版

◎ 江红 余青松 编著

- 教材 + 练习册 + 上机指导 + 微课视频
- 基于Windows 10+Python 3.7
- 700多个实例，431道复习题，563个实践操作任务，37个综合应用案例

《教学课件》《教学大纲》《电子教案》《程序源码》《期末试卷》《习题答案》

清华大学出版社

21世纪高等学校计算机类课程创新规划教材·微课版

Python程序设计与算法基础教程

（第2版） 微课版

江 红 余青松 编著

清华大学出版社
北 京

内 容 简 介

本书集教材、练习册、上机指导于一体，基于 Windows 10 和 Python 3.7 构建 Python 开发平台，阐述 Python 语言的基础知识，以及使用 Python 语言的开发应用实例，具体内容包括 Python 概述，Python 语言基础，程序流程控制，常用内置数据类型，序列数据类型，输入和输出，错误和异常处理，函数、类和对象，模块和客户端，算法与数据结构基础，图形用户界面，图形绘制，数值日期和时间处理，字符串和文本处理，文件、数据库访问，网络和 Web 编程，多线程编程以及系统管理等。

本书编者结合多年的程序设计、系统开发以及授课经验，由浅入深、循序渐进地介绍 Python 程序设计语言，让读者能够较为系统、全面地掌握程序设计的理论和应用。本书还提供了教学微课视频。

本书可以作为高等学校各专业的计算机程序设计教程，也可作为广大程序设计开发者、爱好者的自学参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

Python 程序设计与算法基础教程：微课版/江红，余青松编著. —2 版. —北京：清华大学出版社，2019
(21 世纪高等学校计算机类课程创新规划教材·微课版)
ISBN 978-7-302-52483-0

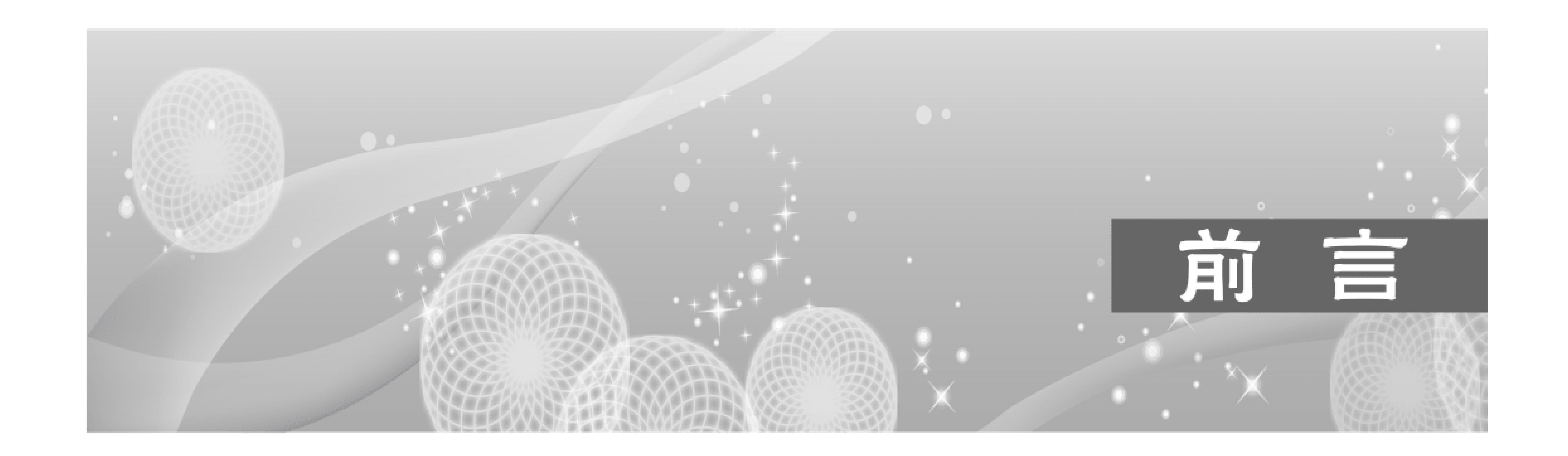
Ⅰ. ①P… Ⅱ. ①江… ②余… Ⅲ. ①软件工具－程序设计－高等学校－教材 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2019)第 043701 号

策划编辑：魏江江
责任编辑：王冰飞
封面设计：刘　键
责任校对：李建庄
责任印制：杨　艳

出版发行：清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦 A 座　**邮　编**：100084
社 总 机：010-62770175　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：清华大学印刷厂
经　销：全国新华书店
开　本：185mm×260mm　**印　张**：26.25　**字　数**：674 千字
版　次：2017 年 7 月第 1 版　2019 年 7 月第 2 版　**印　次**：2019 年 7 月第1 次印刷
印　数：19001～22000
定　价：59.00 元

产品编号：081892-01



前言

程序设计是大专院校计算机、电子信息、工商管理等相关专业的必修课程。Python 语言是一种解释型、面向对象的计算机程序设计语言，广泛用于计算机程序设计教学语言、系统管理编程脚本语言、科学计算等，特别适用于快速的应用程序开发。Python 编程语言广受开发者的喜爱，并被列入 LAMP(Linux、Apache、MySQL 以及 Python/Perl/PHP)，已经成为最受欢迎的程序设计语言之一。

本书集教材、练习册、上机指导于一体，基于 Windows 10 和 Python 3.7 构建 Python 开发平台，通过大量的实例由浅入深、循序渐进地阐述 Python 语言的基础知识，以及使用 Python 语言的开发应用实例，具体内容包括 Python 概述，Python 语言基础，程序流程控制，常用内置数据类型，序列数据类型，输入和输出，错误和异常处理，函数、类和对象，模块和客户端，算法与数据结构基础，图形用户界面，图形绘制，数值日期和时间处理，字符串和文本处理，文件、数据库访问，网络和 Web 编程，多线程编程以及系统管理等。

本书是第 1 版的升级和完善。

在第 1 版的基础上，在每个章节中增加了“蒙特卡洛模拟：赌徒破产命运”“基于字典的通信录”“使用随机数估值圆周率”“去除列表中的重复项生成器函数”“文本统计”“基因预测”“字符串加密和解密”“病毒扫描”“遍历并输出文件目录结构”等实用小案例。

本书的每个章节末还增加了“网络爬虫案例”“百度音乐批量下载器”“使用 pandas 进行数据分析和处理”“猜单词游戏”“井字棋(Tic Tac Toe)游戏”“21 点扑克牌游戏”“简易图形用户界面计算器”“基于 turtle 的汉诺塔问题求解动画的设计和实现”“基于模块的库存管理系统”“基于数据库和 GUI 的教务管理系统”“文本相似度比较分析”“文本统计并行处理”“科学计算和数据分析”“使用嵌套循环实现图像处理算法”“NLTK 与自然语言处理”等大的实用案例研究。实用案例研究作为本书的电子资源，采用二维码的方式印在书上，作为开源的补充阅读和学习资源，并且随着 Python 程序的需求和演变将不断增补和更新。

教程还提供教学微课视频，方便学生反复观看和学习课程相关内容，扫描书中的二维码，可以在线观看视频讲解。

为了更好地帮助读者理解和掌握知识点及应用技能，本书提供了 700 多个大大小小的实例、431 道复习题(选择题、填空题和思考题)、563 个实践操作任务、37 个综合应用案例。本书

配套的教学课件、教学大纲、电子教案、期末试卷、习题答案可以通过扫描封底课件二维码下载。

本书由华东师范大学江红和余青松共同编写，衷心感谢清华大学出版社的编辑，敬佩他们的睿智和敬业。由于时间和编者学识有限，书中不足之处在所难免，敬请诸位同行、专家和读者指正。

编　者

2019 年 5 月





源码下载

第 1 章

Python 概述


视频讲解

Python 语言是一种解释型、面向对象的计算机程序设计语言。Python 语言广泛用于计算机程序设计教学语言、系统管理编程脚本语言、科学计算等，特别适用于快速的应用程序开发。

1.1 初识 Python 语言

1.1.1 Python 语言简介

Python 语言是一种解释型、面向对象的编程语言，由吉多·范罗苏姆(Guido van Rossum)于 1989 年底发明，被广泛应用于处理系统管理任务和科学计算。

Python 是一种开源语言，拥有大量的库，可以高效地开发各种应用程序。

1.1.2 Python 语言的特点

Python 语言具有下列特点。

(1) 简单：Python 是一种解释型的编程语言，遵循优雅、明确、简单的设计哲学，语法简单，易学、易读、易维护。

(2) 高级：Python 属于高级语言，编程者无须考虑底层细节(例如内存分配和释放等)。Python 还包括了内置的高级数据结构(例如 list 和 dict)。

(3) 面向对象：Python 既支持面向过程的编程又支持面向对象的编程，Python 还支持继承、重载，有利于源代码的复用性。

(4) 可扩展性(Extensible)：Python 提供了丰富的 API 和工具，以便程序员能够轻松地使用 C、C++语言来编写扩充模块。

(5) 免费和开源：Python 是 FLOSS(自由/开放源码软件)之一，允许开发者自由地发布此软件的副本、阅读和修改其源代码、将其一部分用于新的自由软件中。

(6) 可移植性：基于其开源本质，Python 已经被移植到许多平台上，包括 Linux/UNIX、Windows、Macintosh 等。用户编写的 Python 程序，如果未使用依赖于系统的特性，无须修改就可以在任何支持 Python 的平台上运行。

(7) 丰富的库：Python 语言提供了功能丰富的标准库，包括正则表达式、文档生成、单元测试、数据库、GUI(图形用户界面)等，还有许多其他高质量的库，例如 Python 图像库等。

(8) 可嵌入性：用户可以将 Python 嵌入到 C、C++程序，从而为 C、C++程序提供脚本功能。

1.1.3 Python 语言的应用范围

Python 具有广泛的应用范围，常用的应用场景如下。

（1）操作系统管理：Python 作为一种解释型的脚本语言，特别适合于编写操作系统管理脚本，使用 Python 编写的系统管理脚本在可读性、源代码重用度、扩展性等方面都优于普通的 shell 脚本。

（2）科学计算：Python 程序员可以使用 NumPy、SciPy、Matplotlib 等模块编写科学计算程序。众多开源的科学计算软件包均提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV、三维可视化库 VTK、医学图像处理库 ITK 等。

（3）Web 应用：Python 经常被用于 Web 开发，例如通过 mod_wsgi 模块 Apache 可以运行用 Python 编写的 Web 程序。

（4）图形用户界面（GUI）开发：Python 支持 GUI 开发，使用 Tkinter、wxPython 或者 PyQt 库可以开发跨平台的桌面软件。

（5）其他：例如游戏开发，很多游戏使用 C++编写图形显示等高性能模块，而使用 Python 编写游戏的逻辑。

1.2 Python 语言版本和开发环境

1.2.1 Python 语言的版本

Python 目前包含两个主要版本，即 Python 2 和 Python 3。

Python 2.0 于 2000 年 10 月发布，最新版本为 Python 2.7。Python 2 实现了完整的垃圾回收，并且支持 Unicode。目前存在大量使用 Python 2 开发的程序和库。

Python 3.0 于 2008 年 12 月发布。相对于 Python 的早期版本，Python 3 是一个较大的升级。Python 3 在设计时为了不带入过多的累赘，没有考虑向下兼容。

例如，在 Python 3 中不支持 print，而使用新增的 print()函数：

```
print('abc')                # Python 3 正确，Python 2 错误
print 'abc '                # Python 3 错误，Python 2 正确
```

因此，许多针对早期 Python 版本设计的程序都无法在 Python 3 上正常运行。注意，使用 Python 3 一般不能直接调用使用 Python 2 开发的库，而必须使用相应的 Python 3 版本的库。

Python 3 的很多新特性后来被移植到 Python 2.6/2.7 上。作为一个过渡版本，Python 2.6/2.7 基本使用 Python 2.x 的语法和库，也允许使用 Python 3.0 的部分语法和函数。如果程序可以在 Python 2.6/2.7 上正常运行，则可以通过一个名为 2to3 的转换工具（Python 自带的实用脚本）无缝迁移到 Python 3.0 上。

1.2.2 Python 语言的实现

Python 2 和 Python 3 规定相应版本 Python 的语法规则。实现 Python 语法的解释程序就是 Python 的解释器。

Python 解释器用于解释和执行 Python 语句和程序。常用的 Python 实现如下。

（1）CPython：使用 C 语言实现的 Python，即原始的 Python 实现。这是最常用的 Python 版

本，也称之为 ClassicPython。通常 Python 就是指 CPython，当需要区别的时候才使用 CPython。

(2) Jython：使用 Java 语言实现的 Python，原名为 JPython。Jython 可以直接调用 Java 的类库，适用于 Java 平台的开发。

(3) IronPython：面向.NET 的 Python 实现。IronPython 能够直接调用.NET 平台的类，适用于.NET 平台的开发。

(4) PyPy：使用 Python 语言实现的 Python。

1.2.3 Python 语言的集成开发环境

Python 是一种跨平台的脚本语言，在不同平台上提供了众多的集成开发环境(IDE)，可以提高用户的编程效率。常用的集成开发环境如下。

(1) IDLE：Python 内置的集成开发工具。

(2) Spyder：使用 Python 编程语言进行科学计算的集成开发环境。

(3) PyCharm：由 JetBrains 公司开发的商业 Python IDE，支持企业级的开发。

(4) Eclipse+Pydev 插件：在通用集成开发环境 Eclipse 上安装 Pydev 插件，可以实现 Python 集成开发环境，方便调试程序。

(5) Visual Studio+Python Tools for Visual Studio：在 Visual Studio 基础上安装 Python Tools for Visual Studio，可以使用功能完善的 Visual Studio 开发 Python 程序。

(6) PythonWin：适用于 Windows 环境下的 Python 集成开发工具。

1.3 下载和安装 Python

1.3.1 下载 Python

Python 支持多平台，不同平台的安装和配置大致相同。本书基于 Windows 10 和 Python 3.7 构建 Python 开发平台。

【例 1.1】 下载 Python 安装程序。

(1) 打开 Python 官网 Windows 环境下载页面。在浏览器地址栏中输入"https://www.python.org/downloads/windows/"，按 Enter 键，如图 1-1 所示。

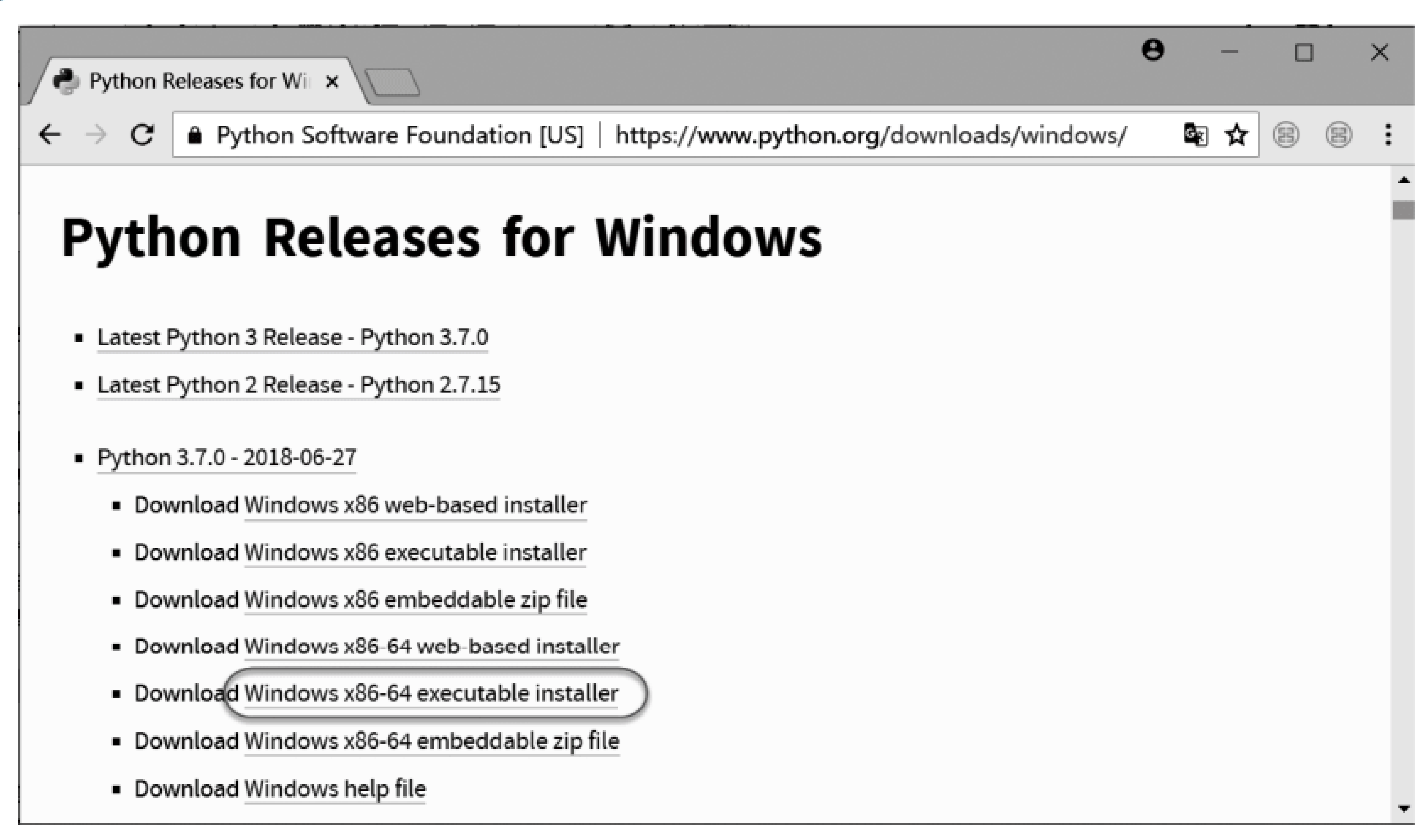


图 1-1 下载 Python

（2）下载 Python 安装程序。单击图 1-1 中的 Windows x86-64 executable installer 超链接，下载目前最新版本 Python 3.7.0（64 位）的安装程序 python-3.7.0.exe（25MB）。

1.3.2 安装 Python

Python 的安装过程与其他 Windows 安装程序类似。

【例 1.2】 安装 Python 应用程序。

（1）运行 Python 安装程序。双击下载的 Windows 格式的安装文件 python-3.7.0-amd64.exe，打开安装程序向导。

（2）设定安装选项。根据安装向导安装 Python，在定制 Python 的对话框中注意选中“Add Python 3.7 to PATH”复选框，如图 1-2 所示。

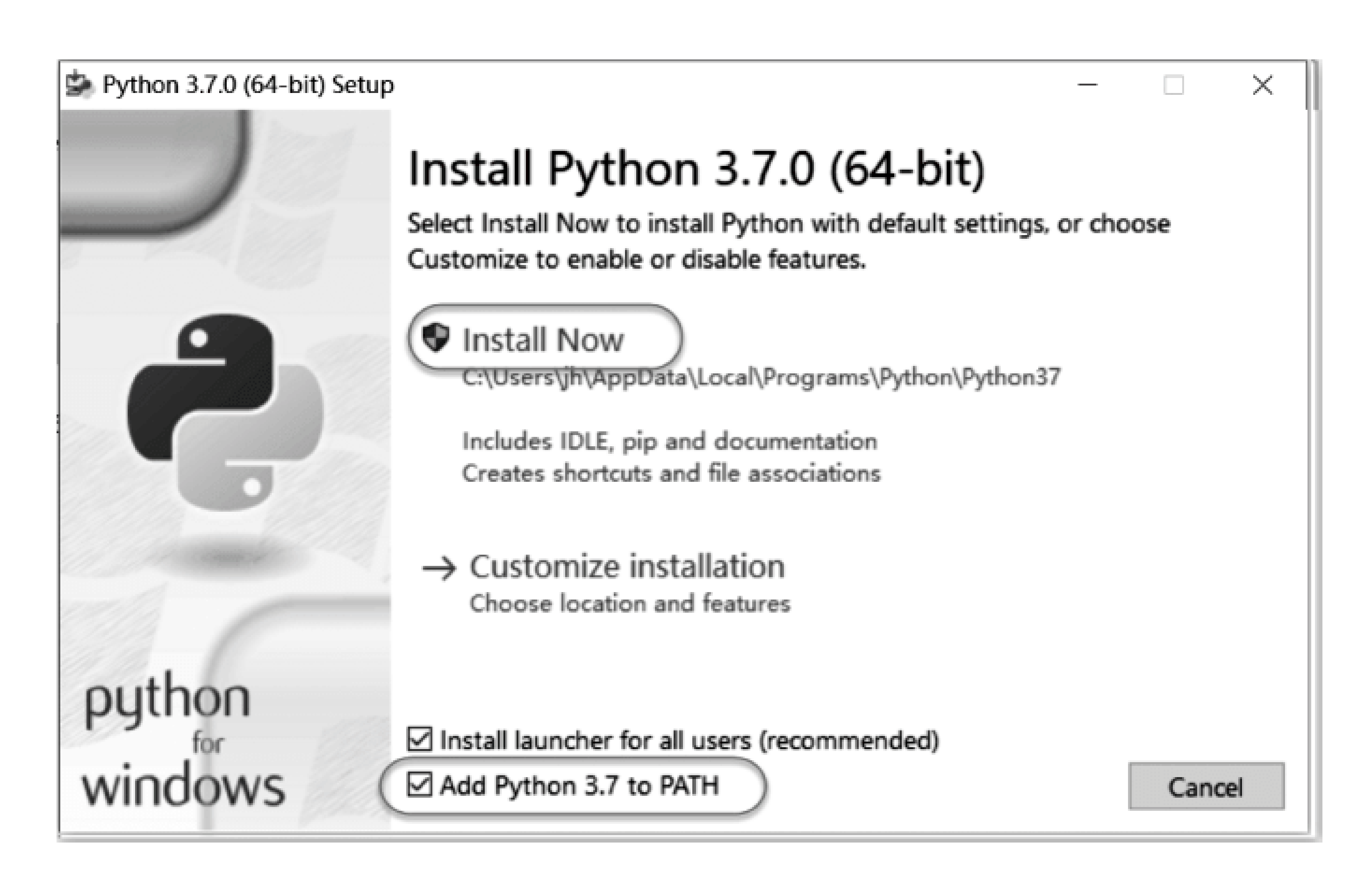


图 1-2 设定 Python 安装选项

（3）安装程序。单击 Install Now 超链接，安装 Python 程序。

1.3.3 安装和管理 Python 扩展包

Python 3.4 以后的版本包含 pip 和 setuptools 库。pip 用于安装和管理 Python 扩展包；setuptools 用于发布 Python 包。

在使用 pip 和 setuptools 前建议先更新到其最新版本。

pip 的典型应用是从 PyPI（Python Package Index）上安装 Python 第三方包。其命令行的基本语法如下。

（1）安装包的最新版本（例如 SomeProject 的最新版本）。

```
python -m pip install SomeProject
```

（2）安装包的某个版本。

```
python -m pip install SomeProject==1.4
```

（3）安装包的某个范围的版本（例如 SomeProject 的大于等于 1 小于 2 的版本）。

```
python -m pip install SomeProject>=1,<2
```

（4）安装包的某个兼容版本（例如 SomeProject 的兼容 1.4.2 的版本）。

```
python -m pip install SomeProject~=1.4.2
```

（5）更新安装包（例如更新 SomeProject 到最新版本）。

```
python －m pip install －U SomeProject
```

说明：

（1）在 Python 的安装目录“Python37\Scripts”中还包含 pip. exe、pip3. exe、pip3. 7. exe，它们与上述基于 pip 模块安装包等价。例如可以使用命令行安装包：

```
C:\Users\jh> pip install numpy
```

（2）如果安装包时 Python 产生错误“[WinError 5]拒绝访问”，可以使用管理员权限打开命令提示符窗口进行安装，或使用-user 安装到个人目录中。

【例 1.3】 更新 pip 和 setuptools 包。

在 Windows 命令提示符窗口中输入命令行命令“python -m pip install -U pip setuptools”，以更新 pip 和 setuptools 包，如图 1-3 所示。

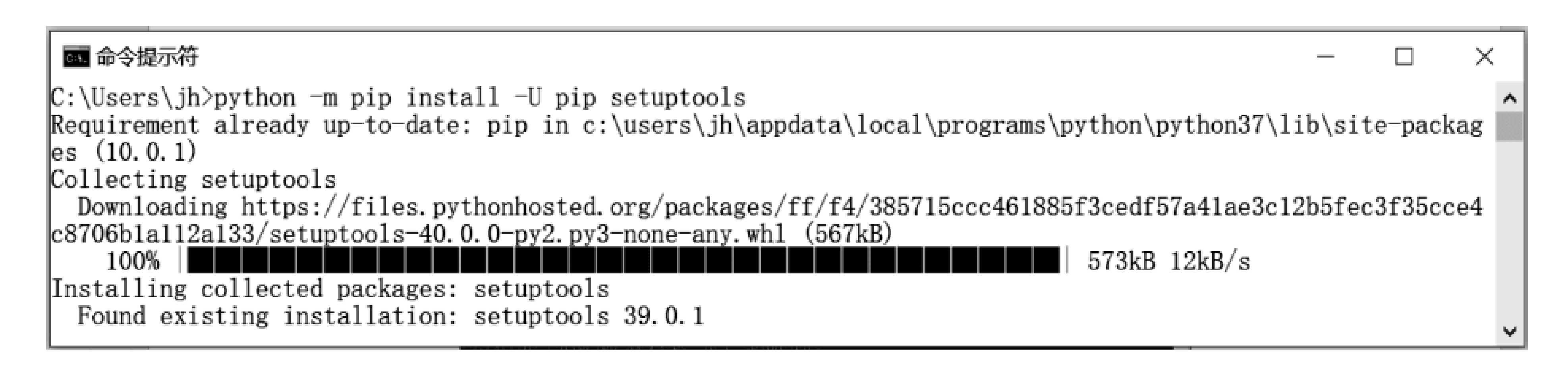


图 1-3　更新 pip 和 setuptools 包

【例 1.4】 安装 NumPy 包。Python 扩展模块 NumPy 提供了数组和矩阵处理，以及傅立叶变换等高效的数值处理功能。

在 Windows 命令提示符窗口中输入命令行命令“python -m pip install NumPy”，以安装 NumPy 包，如图 1-4 所示。

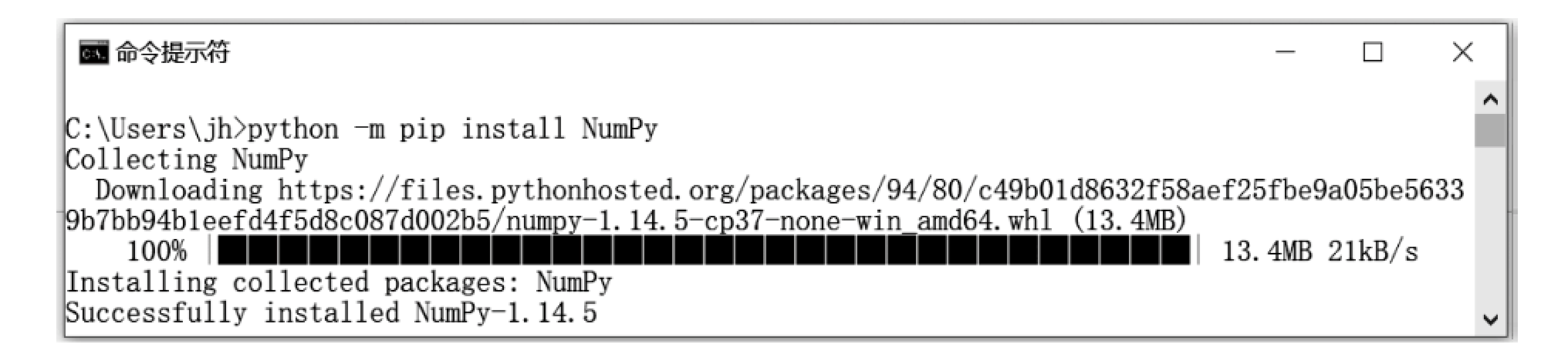


图 1-4　安装 NumPy 包

【例 1.5】 安装 Matplotlib 包。Matplotlib 是 Python 最著名的绘图库之一，提供了一整套和 MATLAB 相似的命令 API，既适合交互式地进行制图，也可以作为绘图控件方便地嵌入 GUI 应用程序中。Matplotlib 的具体应用将在本书第 13 章中详细介绍。

在 Windows 命令提示符窗口中输入命令行命令“python -m pip install Matplotlib”，以安装 Matplotlib 包，如图 1-5 所示。

```
命令提示符 - python -m pip install Matplotlib
C:\Users\jh>python -m pip install Matplotlib
Collecting Matplotlib
  Downloading https://files.pythonhosted.org/packages/ec/ed/46b835da53b7ed05bd4c6cae293f13ec26e877d2
e490a53a709915a9dcb7/matplotlib-2.2.2.tar.gz (37.3MB)
    1% |▌                               | 450kB 12kB/s eta 0:49:41
```

图 1-5　安装 Matplotlib 包

1.4 使用 Python 解释器解释执行 Python 程序

1.4.1 运行 Python 解释器

Python 默认的安装路径为用户本地应用程序文件夹下的 Python 目录（例如“C:\Users\jh\AppData\Local\Programs\Python\Python37”），在该目录下包括 Python 解释器 python.exe，以及 Python 库目录和其他文件。

用户可以使用命令提示符窗口运行 python.exe，也可以通过 Windows 开始菜单运行 python.exe。

注意：在控制台上交互式地执行 Python 代码的过程一般称为 REPL（Read-Eval-Print-Loop）。它是学习 Python 语言的重要组成部分，读者可以使用它学习 Python 的基本语法，运行试验新的库函数功能。

【例 1.6】 运行 Python 解释器。

单击“开始”按钮，选择“所有应用”|Python 3.7|Python 3.7 (64-bit)命令，打开 Python 解释器交互窗口，如图 1-6 所示。

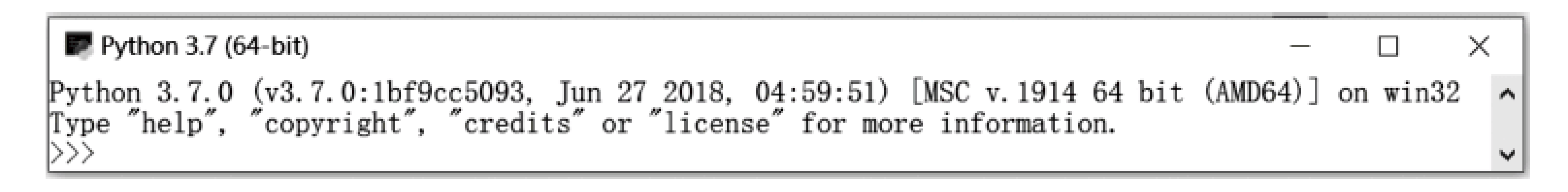


图 1-6 Python 解释器交互窗口

【例 1.7】 输出“Hello，world!”。

Python 解释器的提示符为>>>。在提示符下输入语句，Python 解释器将解释执行，并输出结果。例如输入 print('Hello，world!')，则 Python 解释器将调用 print()函数，打印输出字符串“Hello，world!”，如图 1-7 所示。

【例 1.8】 使用 Python 解释器进行数学运算。

在 Python 解释器的提示符下输入数学公式，Python 解释器将解释执行，实现计算器的功能。例如 11+22+33+44+55，计算结果为 165；$(1+0.01)^{365}$，计算结果为 37.78343433288728，如图 1-8 所示。

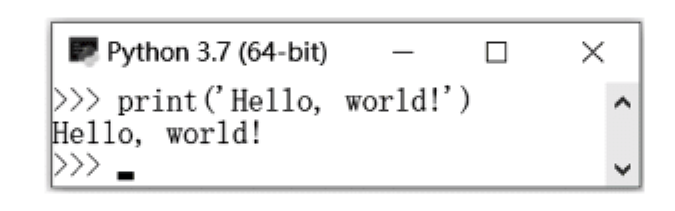


图 1-7 Python 解释器输出“Hello，world!”

```
Python 3.7 (64-bit)
>>> 11+22+33+44+55
165
>>> (1+0.01)**365
37.78343433288728
>>>
```

图 1-8 使用 Python 解释器进行数学运算

【例 1.9】 使用解释器环境中的特殊变量“_”。

在 Python 解释器环境中存在一个特殊变量“_”，用于表示上一次运算的结果。例如：

```
>>> 11 + 22                    #输出:33
>>> _                          #输出:33
>>> _ + 33                     #输出:66
```

【例 1.10】 同时运行多个表达式。

用户可以同时运行多个以逗号分隔的表达式，返回结果为元组。例如：

```
>>> 2,2 * * 10                    #输出:(2, 1024)
```

【例 1.11】　关闭 Python 解释器。

通过按 Ctrl+Z 组合键及 Enter 键，或者输入 quit()，或者直接关闭 Python 解释器交互窗口，均可以关闭 Python 解释器。

1.4.2　运行 Python 集成开发环境

Python 内置了集成开发环境 IDLE(Integrated DeveLopment Environment 或者 Integrated Development and Learning Environment)。相对于 Python 解释器交互窗口，集成开发环境 IDLE 提供了图形开发用户界面，可以提高 Python 程序的编写效率。

【例 1.12】　运行 Python 内置的集成开发环境 IDLE。

单击"开始"按钮，选择"所有应用"| Python 3.7 | IDLE (Python 3.7 64-bit)命令，打开 Python 内置的集成开发环境 IDLE，如图 1-9 所示。

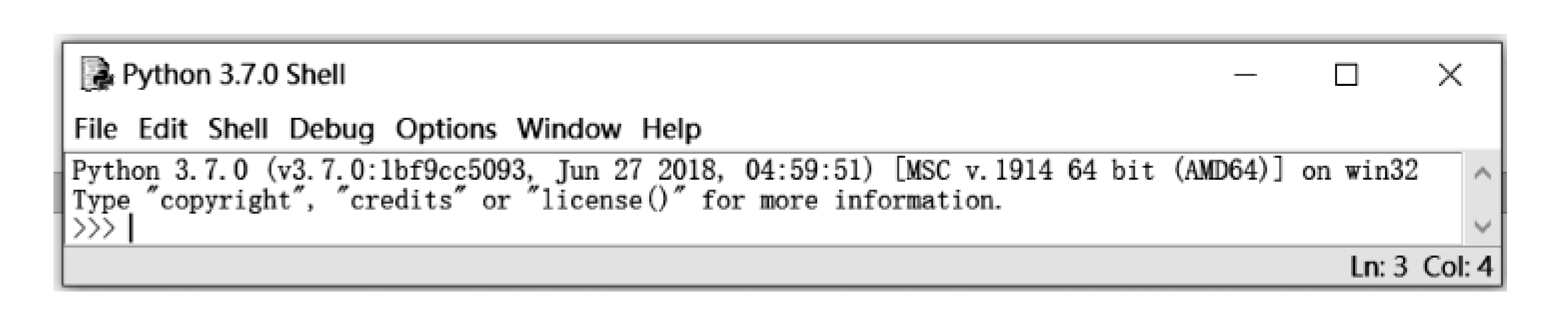


图 1-9　Python 内置的集成开发环境 IDLE

【例 1.13】　使用集成开发环境 IDLE 解释执行 Python 语句。

在 Python 集成开发环境 IDLE 中输入 print('Good!' * 5)，则打印输出字符串"Good! Good! Good! Good! Good!"。注意，print('Good!' * 5)的结果为打印输出 5 个"Good!"的拼接，如图 1-10 所示。

【例 1.14】　使用 IDLE 执行多行代码。

复杂的 Python 语句包含多行代码。例如，以下循环语句用于打印 0～9 的数字，分隔符为空格：

```
for x in range(10):
    print(x, end = ' ')
```

在 Python 解释器的提示符下输入"for x in range(10):"后(注：冒号代表复合语句)，按 Enter 键，Python 解释器将在下一行自动缩进，等待输入；输入 print(x, end=' ')后，按 Enter 键，Python 解释器将在下一行等待输入(注：for 循环语句块可以包含多条语句)。直接按 Enter 键(本例中的 for 循环语句块只包含一条语句)，结束 for 循环语句，Python 解释器解释执行各语句并输出结果，如图 1-11 所示。

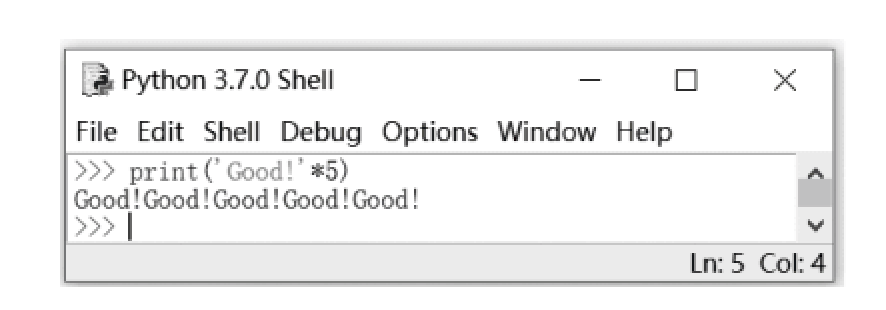


图 1-10　使用 IDLE 解释执行 Python 语句

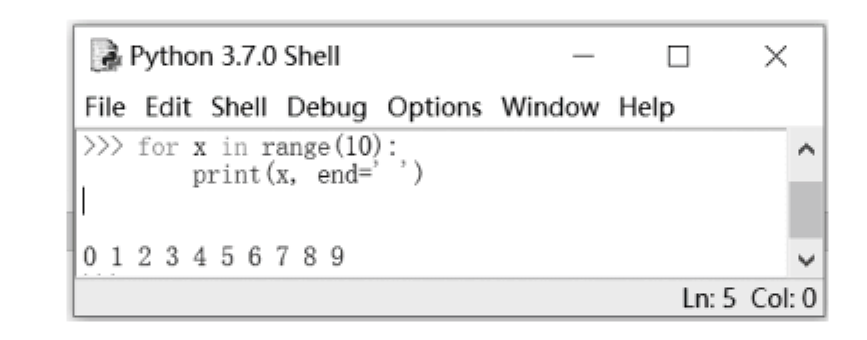


图 1-11　使用 Python 解释器执行多行代码

【例 1.15】　关闭 Python 解释器。

输入 quit()，或者直接关闭 IDLE 窗口，均可以关闭 Python 解释器。

1.5 使用文本编辑器和命令行编写和执行 Python 源文件程序

Python 解释器命令行采用交互方式执行 Python 语句，其优点是方便、直接，但是在交互式环境下需要逐条输入语句，且执行的语句没有保存到文件中，因而不能重复执行，故不适合于复杂规模的程序设计。

用户可以把 Python 程序编写成一个文本文件，其扩展名通常为.py，然后通过 Python 解释器编译执行。

使用文本编辑器和命令行编写和执行 Python 源文件程序的过程包括以下 3 个步骤。

（1）创建 Python 源代码文件，即扩展名为.py 的文件，例如 hello.py。

（2）把 Python 源代码程序文件编译成字节码程序文件，即扩展名为.pyc 的文件，例如 hello.pyc。Python 的编译是一个自动过程，用户一般不必在意它的存在。编译成字节码可以节省加载模块的时间，提高效率。

（3）加载并解释执行 Python 程序。

编写 Python 源代码文件程序并通过 Python 编译器/解释器执行程序的流程如图 1-12 所示（以 hello.py 为例）。

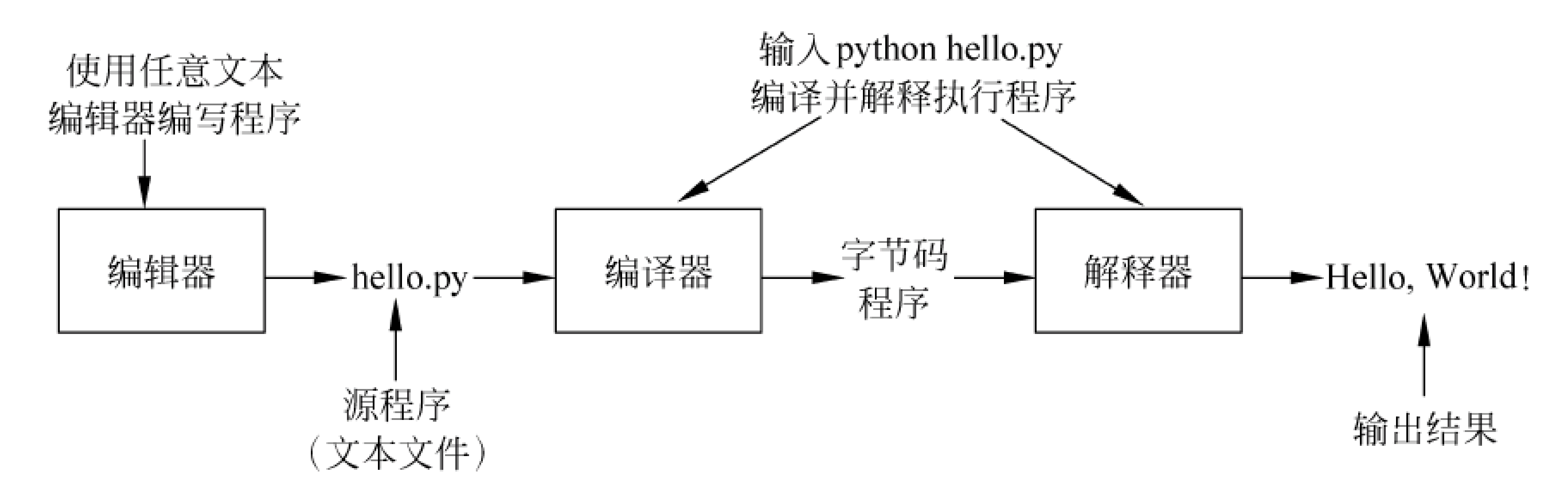


图 1-12 编写、编译和执行 Python 程序

1.5.1 编写输出“Hello，World!”的程序

使用文本编辑软件（例如 Windows 记事本 Notepad.exe）在“C:\pythonpa\ch01”目录下创建程序文件 hello.py。

准备工作：创建用于保存源文件的目录。打开资源管理器，在 C 盘根目录中创建子目录 pythonpa，然后在“C:\pythonpa”下创建子目录 ch01。

注意： 本书正文源代码保存在“C:\pythonpa”中的各章节子目录下，例如第 1 章的源代码保存在“C:\pythonpa\ch01”中，依此类推。

【例 1.16】 使用文本编辑器（记事本）编写输出“Hello，World!”的程序。

（1）运行 Windows 记事本程序。

（2）在记事本中输入程序源代码，如图 1-13 所示。

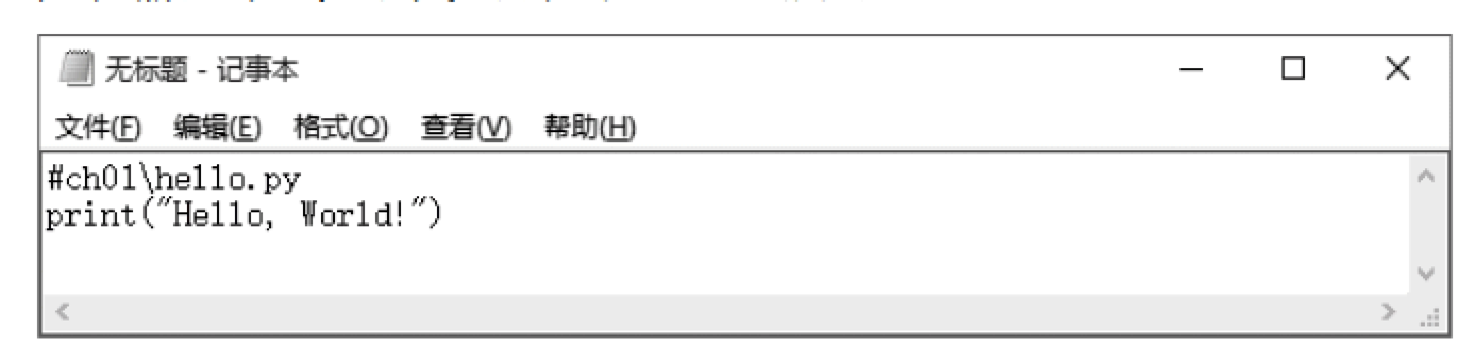


图 1-13 使用文本编辑器（记事本）编写输出“Hello，World!”的程序



(3) 将文件另存为 hello. py。通过选择记事本的“文件”|“另存为”命令,将源程序文件 hello. py 保存到“C:\pythonpa\ch01”中。注意:“保存类型”选择“所有文件”,“编码”选择 UTF-8,如图 1-14 所示。



图 1-14 保存源程序文件 hello. py 到“C:\pythonpa\ch01”中

1.5.2 输出“Hello,World!”程序的源代码分析

第 1 行为注释。Python 注释以符号#开始,到行尾结束。

第 2 行调用内置库的 print()函数,输出“Hello, World!”。

1.5.3 运行 Python 源代码程序

在 Windows 命令提示符窗口中输入命令行命令“python C:\Pythonpa\ch01\hello. py”,直接调用 Python 解释器执行程序 hello. py,并输出结果。

用户也可以在 Windows 命令提示符窗口中输入命令行命令“C:\Pythonpa\ch01\hello. py”,间接调用 Python 解释器执行程序 hello. py,并输出结果。

注意:*在安装 Python 后,Windows 关联扩展名为. py 的文件的默认打开程序为 Python Launcher for Windows(Console)。*

【例 1.17】 使用 Windows 命令提示符窗口运行 hello. py。

(1) 打开 Windows 命令提示符窗口。单击“开始”按钮,选择“所有应用”|“Windows 系统”|“命令提示符”命令,打开 Windows 命令提示符窗口,如图 1-15 所示。

(2) 直接调用 Python 解释器执行程序 hello. py。输入命令行命令“python C:\pythonpa\ch01\hello. py”,按 Enter 键执行程序。

(3) 间接调用 Python 解释器执行程序 hello. py。输入命令行命令“C:\pythonpa\ch01\hello. py”,按 Enter 键执行程序。

(4) 切换到工作目录,即输入“cd C:\pythonpa\ch01”,然后输入命令行命令“python hello. py”,按 Enter 键执行程序。

(5) 切换到工作目录“C:\pythonpa\ch01”,然后输入命令行命令“hello. py”,按 Enter 键执行程序。

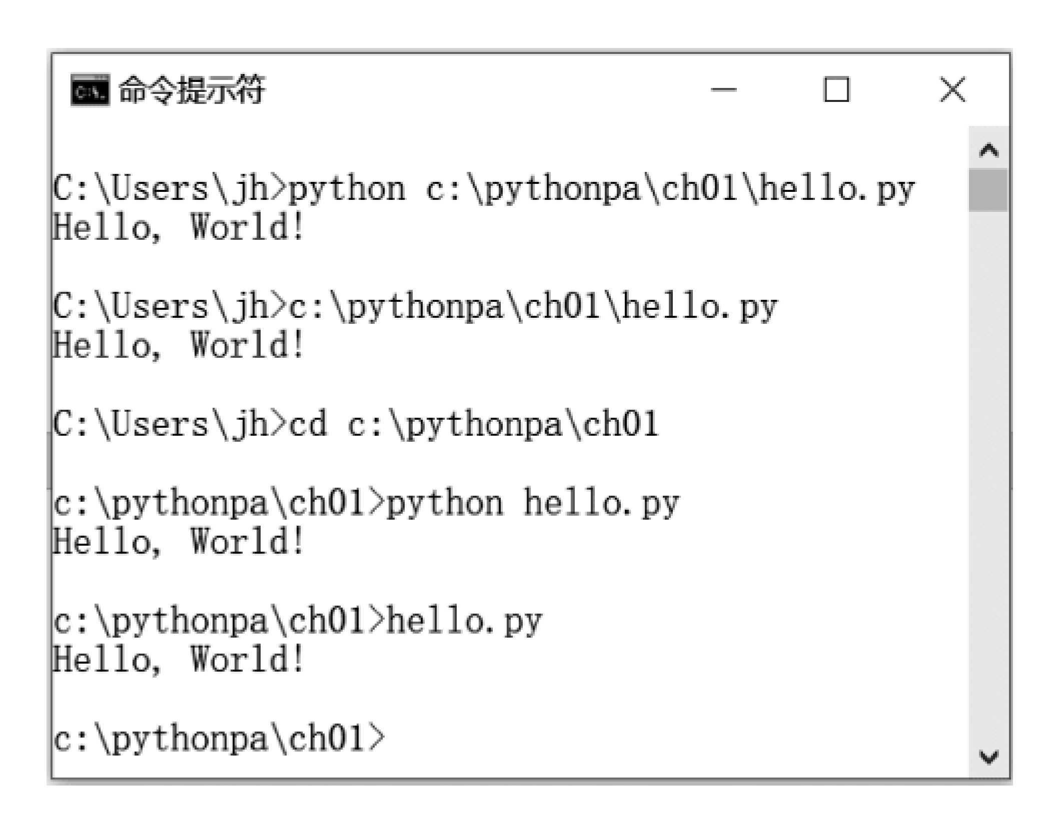


图 1-15　使用 Windows 命令提示符窗口运行 hello.py

【例 1.18】　使用资源管理器运行 hello1.py。

（1）运行 Windows 记事本程序，编写 hello1.py 程序，hello1.py 程序的内容如下。

```
import random                  #导入库模块
print("Hello, World")          #输出:Hello, World
print("你今天的幸运随机数是:", random.choice(range(10)))   #输出从 0 到 9 之间随机选择的数
input()                        #等待用户输入
```

（2）在资源管理器中双击“C:\pythonpa\ch01”目录下的 hello1.py 文件，Windows 自动调用其默认打开程序 Python Launcher for Windows(Console)解释执行 hello1.py 源程序，如图 1-16 所示。

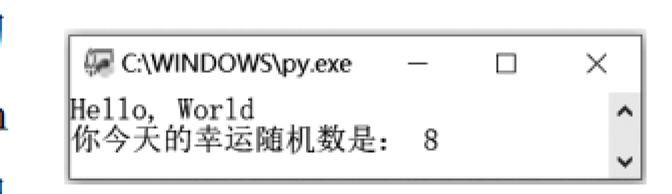


图 1-16　使用资源管理器运行 hello1.py

hello1.py 程序中每一行代码的含义如下。

- 第 1 行代码导入库模块 random。Python 可以导入和使用功能丰富的标准库或扩展库。
- 第 2 行代码调用内置库函数 print()输出“Hello，World”。
- 第 3 行代码使用 random 库中的 choice()函数在 0～9 中随机选择一个数并输出。
- 第 4 行代码调用内置库函数 input()。用户按 Enter 键，程序结束运行。

注意：hello1.py 文件最后包含一个函数 input()，用于等待用户输入，按 Enter 键后，程序结束运行，并关闭窗口。如果不包含该函数，则双击 hello1.py，程序运行后会自动关闭 Windows 命令提示符窗口，从而无法观察到程序运行的结果。

random 是 Python 的标准模块，其具体使用请参见本书第 14 章中的相关内容。

1.5.4　命令行参数

在操作系统命令行运行程序时可以指定若干命令行参数。例如：

```
python c:\test.py Para1 Para2
```

在 Python 程序中导入 sys 模块后可以通过列表 sys.argv 访问命令行参数。argv[0]为 Python 脚本名，例如“c:\test.py”；argv[1]为第 1 个参数，例如 Para1；argv[2]为第 2 个参数，例如 Para2；依此类推。

【例 1.19】　命令行参数示例(hello_argv.py)。在操作系统命令行运行 Python 程序时根据所指定的命令行参数显示输出相应的 Hello 信息。

```
import sys
print('Hello, ' + sys.argv[1])
```

程序运行结果如图 1-17 所示。

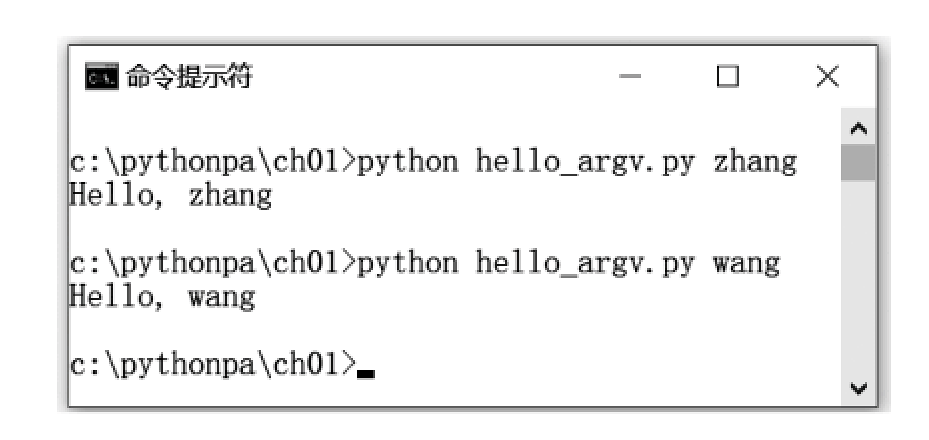


图 1-17 根据命令行参数显示输出

1.6 使用集成开发环境 IDLE 编写和执行 Python 源文件程序

集成开发环境 IDLE 提供了编写和执行 Python 源文件程序的图形界面，可以提高用户 Python 程序的编写效率。

1.6.1 使用 IDLE 编写程序

【例 1.20】 使用 IDLE 编写求解 2 的 1024 次方的程序。

(1) 运行 Python 内置的集成开发环境 IDLE。单击"开始"按钮，选择"所有应用"|Python 3.7|IDLE (Python 3.7 64-bit)命令，打开 Python 内置的集成开发环境 IDLE。

(2) 新建源代码文件。选择 File|New File 命令(或按 Ctrl+N 组合键)，新建 Python 源代码文件，并打开 Python 源代码编辑器。

(3) 输入程序源代码。在 Python 源代码编辑器中输入程序源代码，如图 1-18 所示。

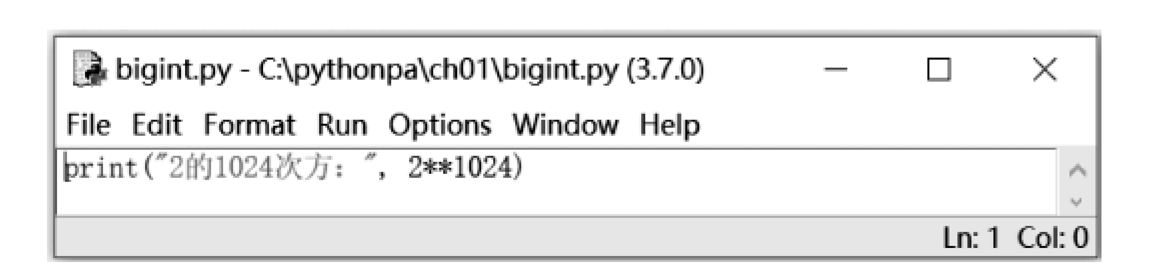


图 1-18 IDLE 源代码编辑器

(4) 将文件保存为 bigint.py。选择 File|Save 命令(或按 Ctrl+S 组合键)，保存文件到位置"C:\pythonpa\ch01"，文件名为 bigint.py。

(5) 运行程序 bigint.py。选择 Run|Run Module 命令(或按 F5 键)，打开 Python 3.7.0 Shell，输出程序的运行结果，如图 1-19 所示。

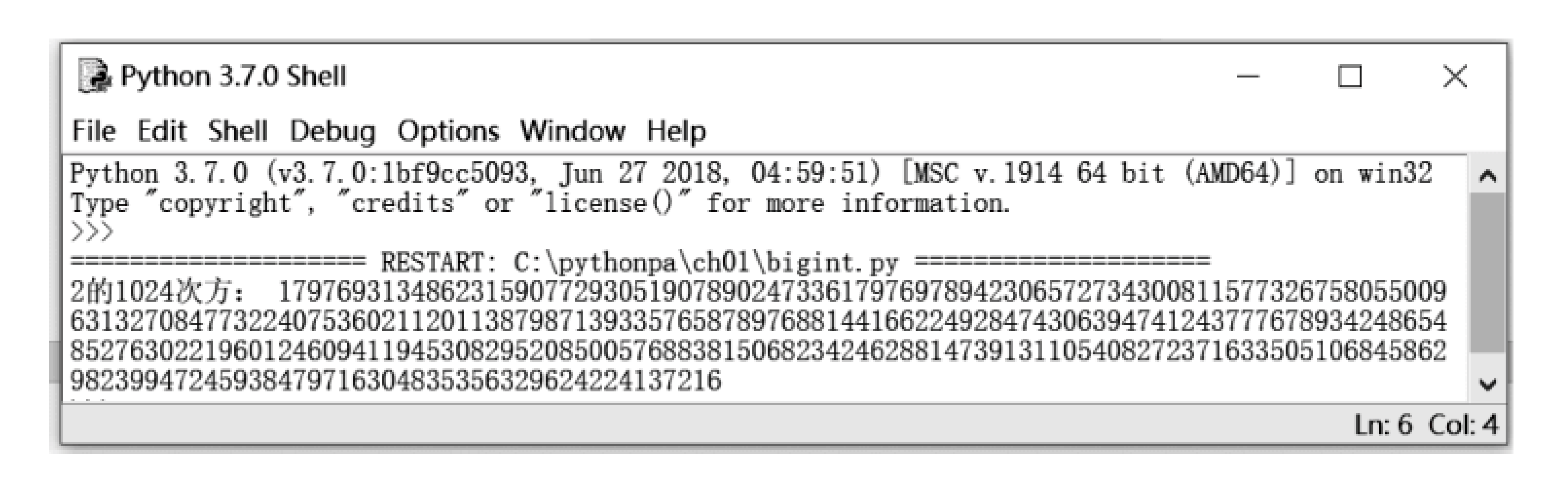


图 1-19 在 IDLE 环境中运行源代码程序

1.6.2 使用 IDLE 编辑程序

【例 1.21】 使用 IDLE 编辑 hello1.py 程序。

（1）运行 Python 内置的集成开发环境 IDLE。

（2）打开程序 hello1.py。按 Ctrl+O 组合键，在随后出现的对话框中选择“C:\pythonpa\ch01\”下的 hello1.py，单击“打开”按钮，打开文件。

（3）编辑文件。在 Python 源代码编辑器中编辑修改程序源代码，将输出“Hello，World”改为输出“Good Luck!”，如图 1-20 所示。

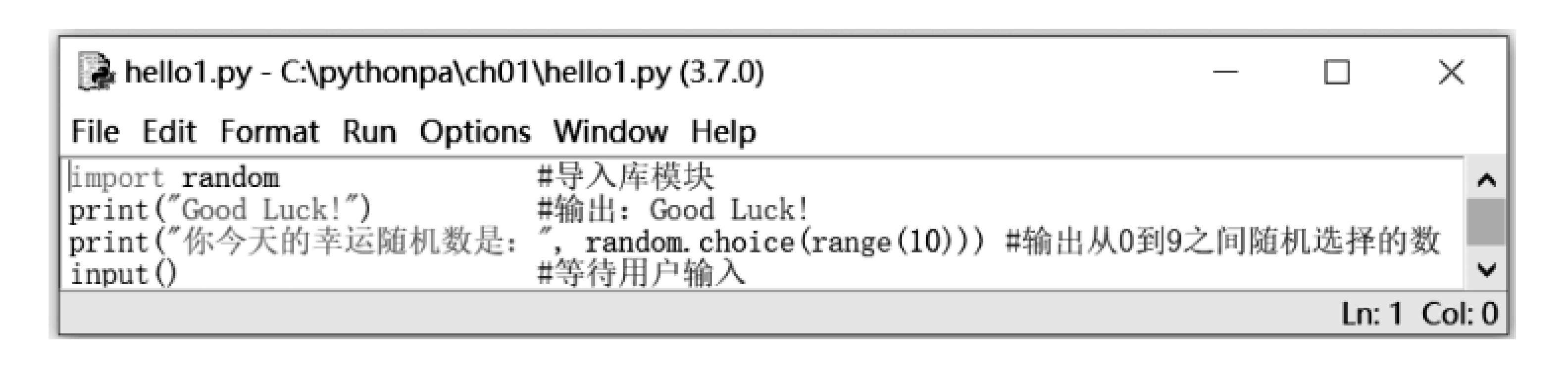


图 1-20 编辑 hello1.py 程序

（4）保存文件 hello1.py。通过按 Ctrl+S 组合键保存文件。

（5）运行程序 hello1.py。通过按 F5 键输出程序的运行结果。

1.7 在线帮助和相关资源

1.7.1 Python 交互式帮助系统

在 Python 中包含了许多内置函数，可以实现交互式帮助，直接输入 help()函数可以进入交互式帮助系统；输入 help(object)可以获取关于 object 对象的帮助信息。

【例 1.22】 使用 Python 交互式帮助系统示例。

（1）进入交互式帮助系统。输入 help()，然后按 Enter 键，如图 1-21 所示。

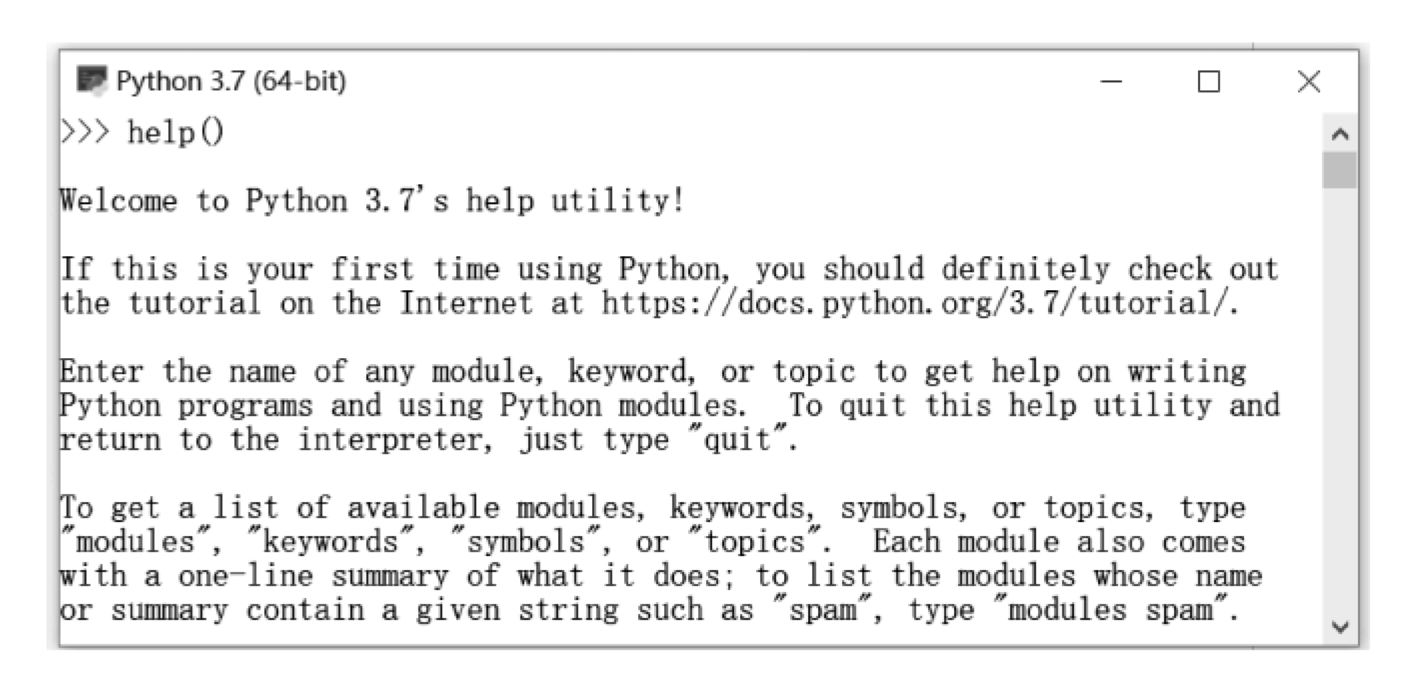
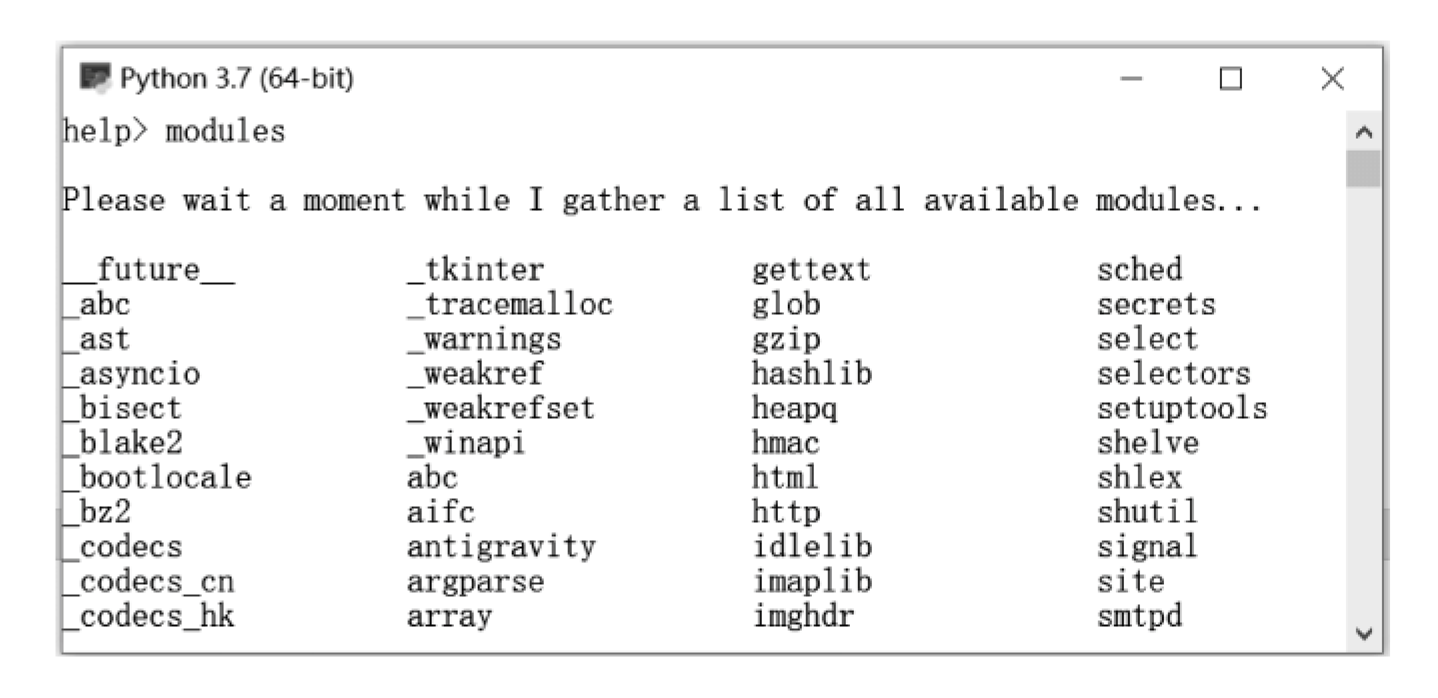


图 1-21 进入交互式帮助系统

（2）显示安装的所有模块。输入 modules，然后按 Enter 键，如图 1-22 所示。

（3）显示与 random 相关的模块。输入 modules random，然后按 Enter 键，如图 1-23 所示。

（4）显示 random 模块的帮助信息。输入 random，然后按 Enter 键，如图 1-24 所示。用户可以通过空格键或者 Enter 键查看下一页帮助信息，通过 Q 或者 q 键结束 random 帮助信息的显示，返回 help 交互式帮助系统界面。

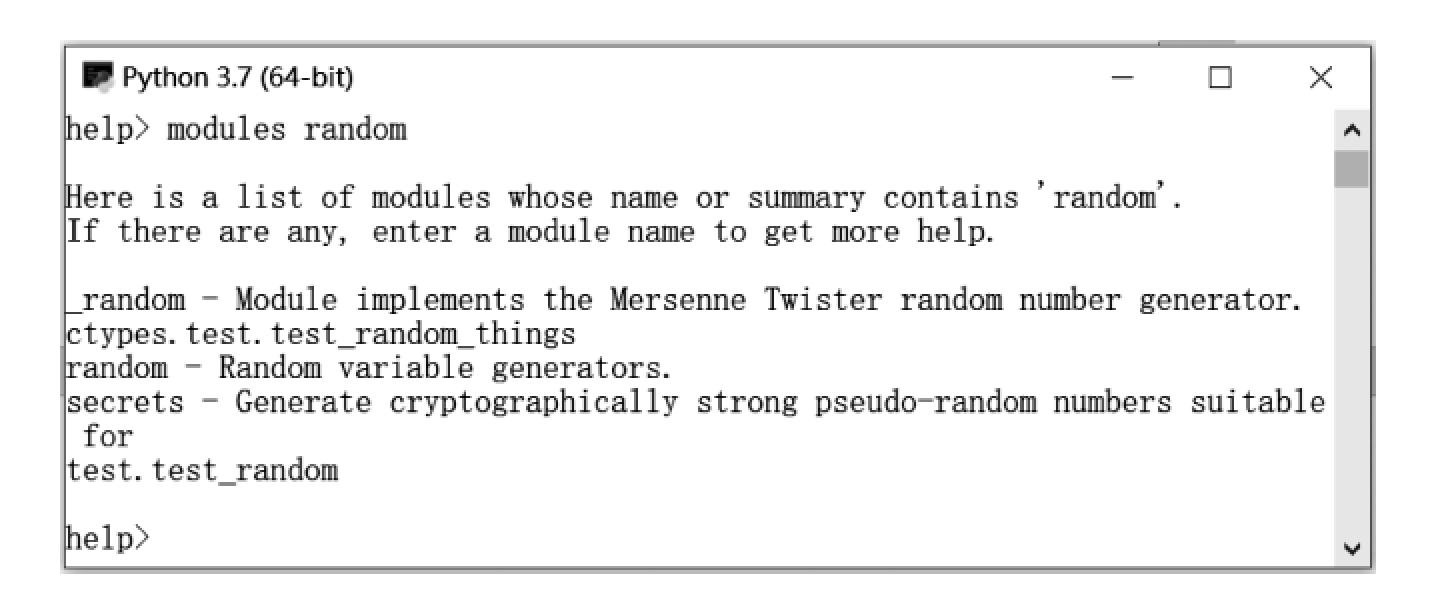


```
help> modules

Please wait a moment while I gather a list of all available modules...

__future__          _tkinter            gettext             sched
_abc                _tracemalloc        glob                secrets
_ast                _warnings           gzip                select
_asyncio            _weakref            hashlib             selectors
_bisect             _weakrefset         heapq               setuptools
_blake2             _winapi             hmac                shelve
_bootlocale         abc                 html                shlex
_bz2                aifc                http                shutil
_codecs             antigravity         idlelib             signal
_codecs_cn          argparse            imaplib             site
_codecs_hk          array               imghdr              smtpd
```

图 1-22 显示安装的所有模块



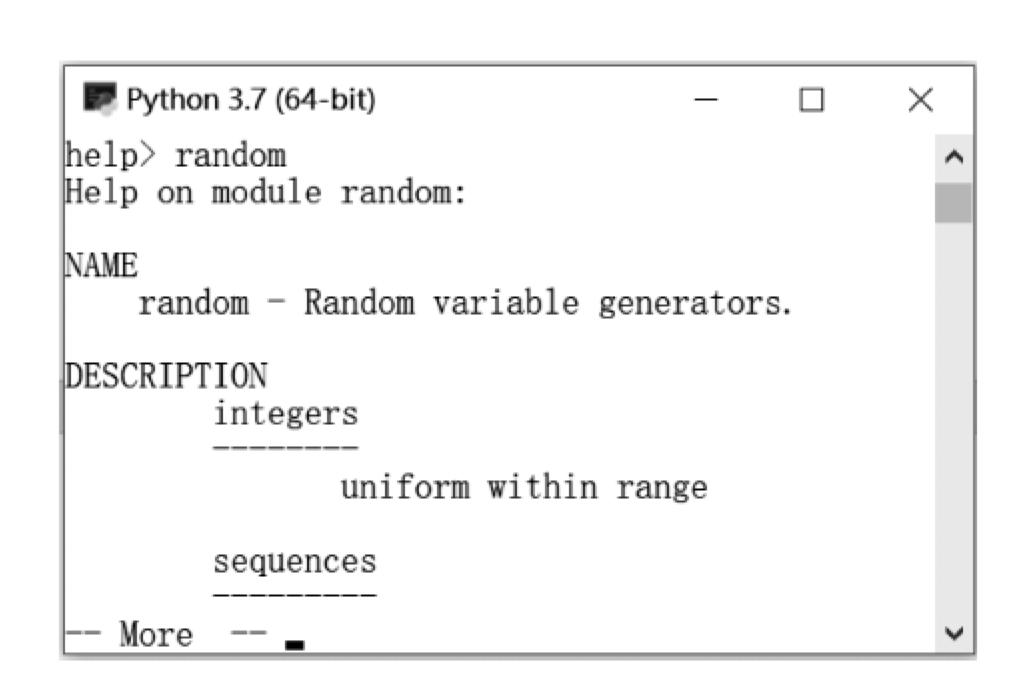
```
help> modules random

Here is a list of modules whose name or summary contains 'random'.
If there are any, enter a module name to get more help.

_random - Module implements the Mersenne Twister random number generator.
ctypes.test.test_random_things
random - Random variable generators.
secrets - Generate cryptographically strong pseudo-random numbers suitable
 for
test.test_random

help>
```

图 1-23 显示与 random 相关的模块



```
help> random
Help on module random:

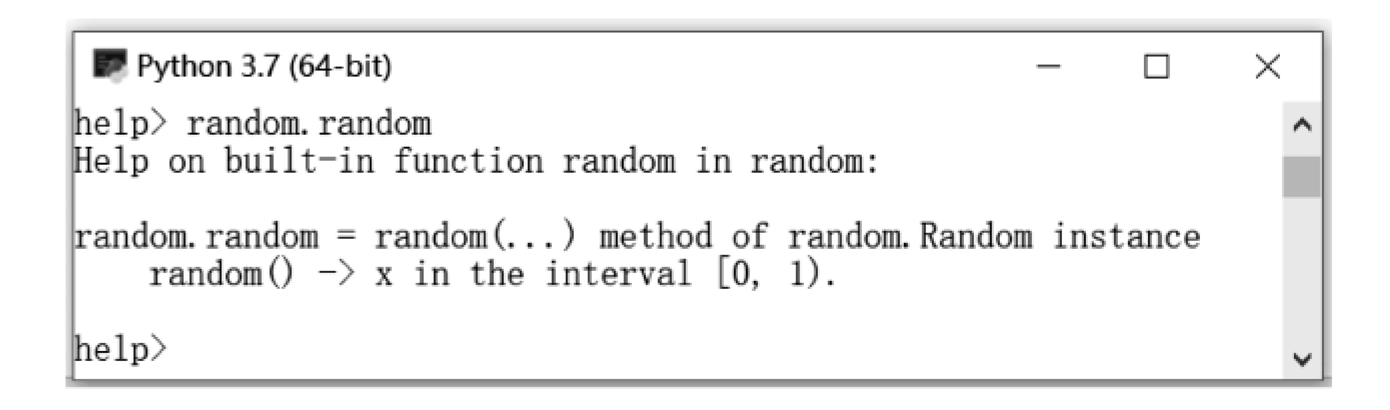
NAME
    random - Random variable generators.

DESCRIPTION
        integers
        --------
               uniform within range

        sequences
        ---------
-- More  --
```

图 1-24 显示 random 模块的帮助信息

(5) 显示 random 模块的 random 函数的信息。输入 random.random,然后按 Enter 键,如图 1-25 所示。



```
help> random.random
Help on built-in function random in random:

random.random = random(...) method of random.Random instance
    random() -> x in the interval [0, 1).

help>
```

图 1-25 显示 random 模块的 random 函数的信息

(6) 退出帮助系统。输入 quit,然后按 Enter 键。

【例 1.23】 使用 Python 内置函数获取帮助信息。

(1) 查看 Python 内置对象列表。输入下列命令:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', …, 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

(2) 查看 float 的信息。输入下列命令:

```
>>> float                        # 输出:<class 'float'>
```

(3) 查看内置类 float 的帮助信息。输入如图 1-26 所示的命令。

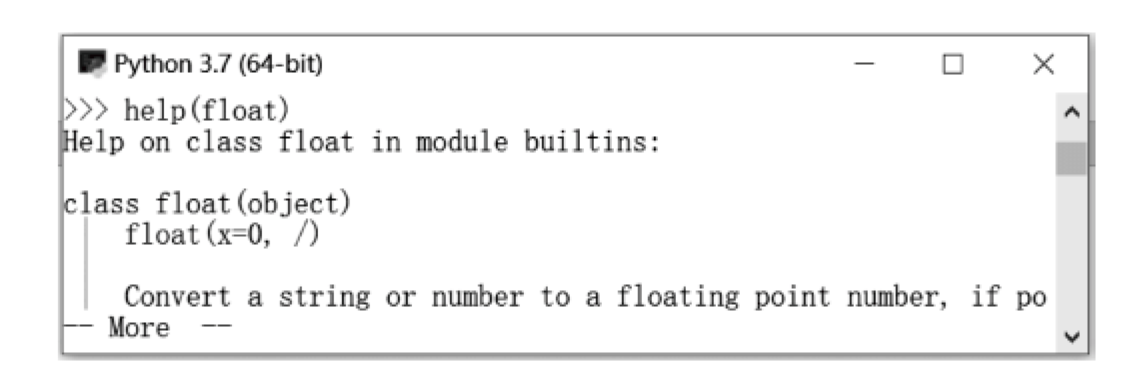


图 1-26 查看内置类 float 的帮助信息

1.7.2 Python 文档

Python 文档提供了有关 Python 语言及标准模块的详细参考信息,是学习和使用 Python 语言编程的不可或缺的工具。

【例 1.24】 使用 Python 文档。

(1) 打开 Python 文档。单击"开始"按钮,选择"所有应用"|Python 3.7|Python 3.7 Manuals (64-bit)命令(用户也可以在 IDLE 环境下按 F1 键),打开 Python 文档,如图 1-27 所示。

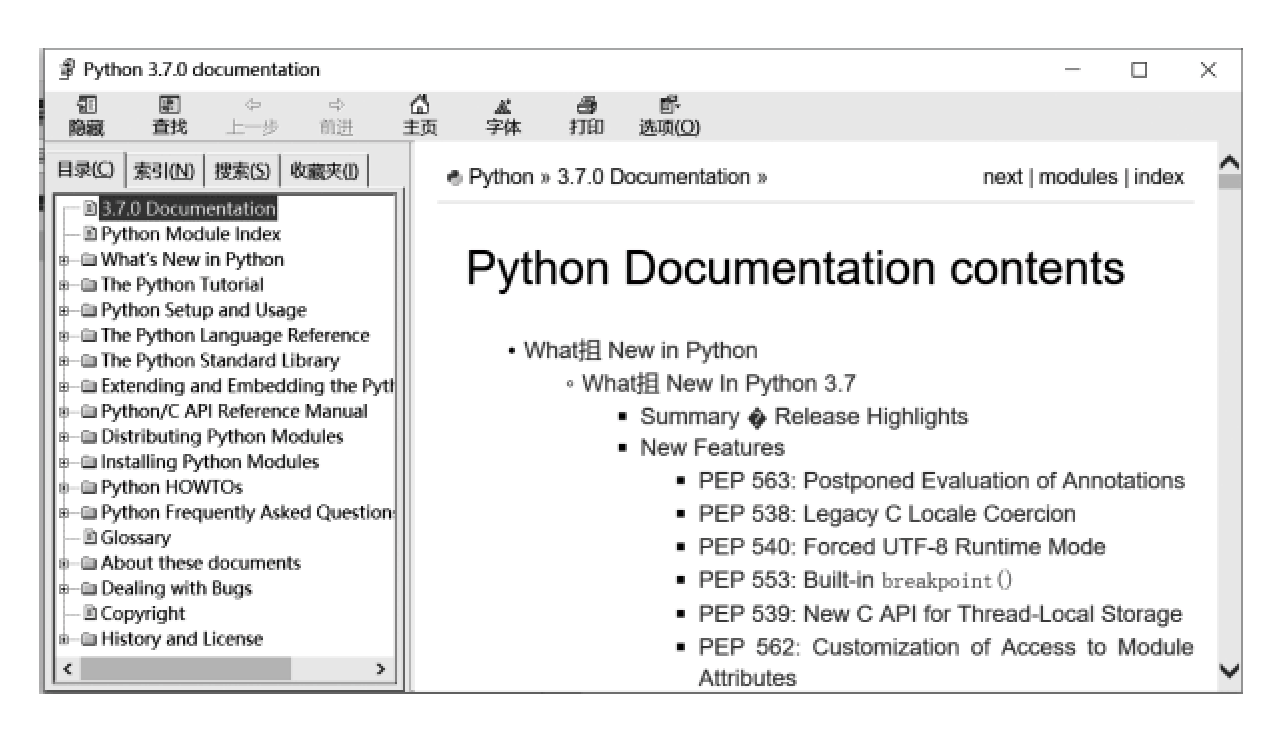


图 1-27 打开 Python 文档

(2) 浏览 random 模块的帮助信息。在左侧的目录树中依次展开,浏览 random 模块的帮助信息,如图 1-28 所示。

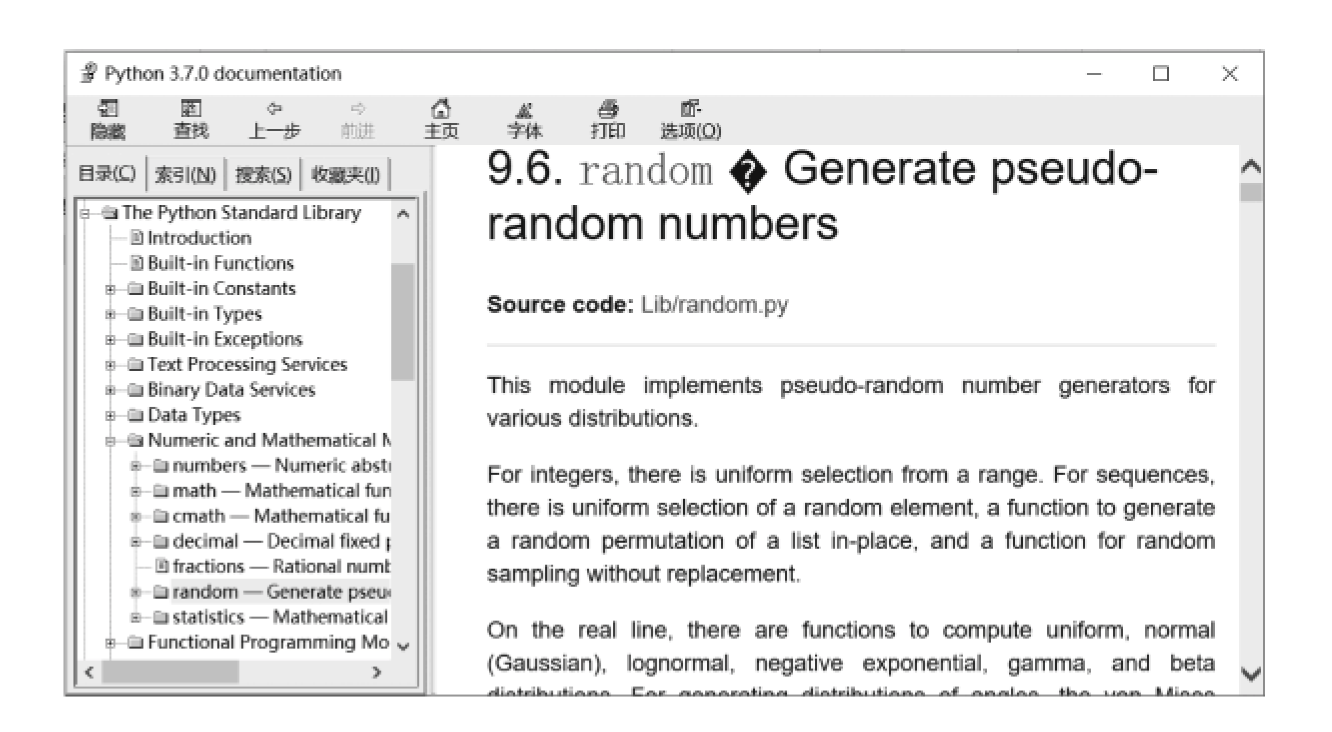


图 1-28 浏览 random 模块的帮助信息

(3) 查找有关 math 的帮助信息。在左侧单击“搜索”选项卡，输入 math，然后按 Enter 键，接着在左侧的目录树中双击查找有关 math 的帮助信息，如图 1-29 所示。

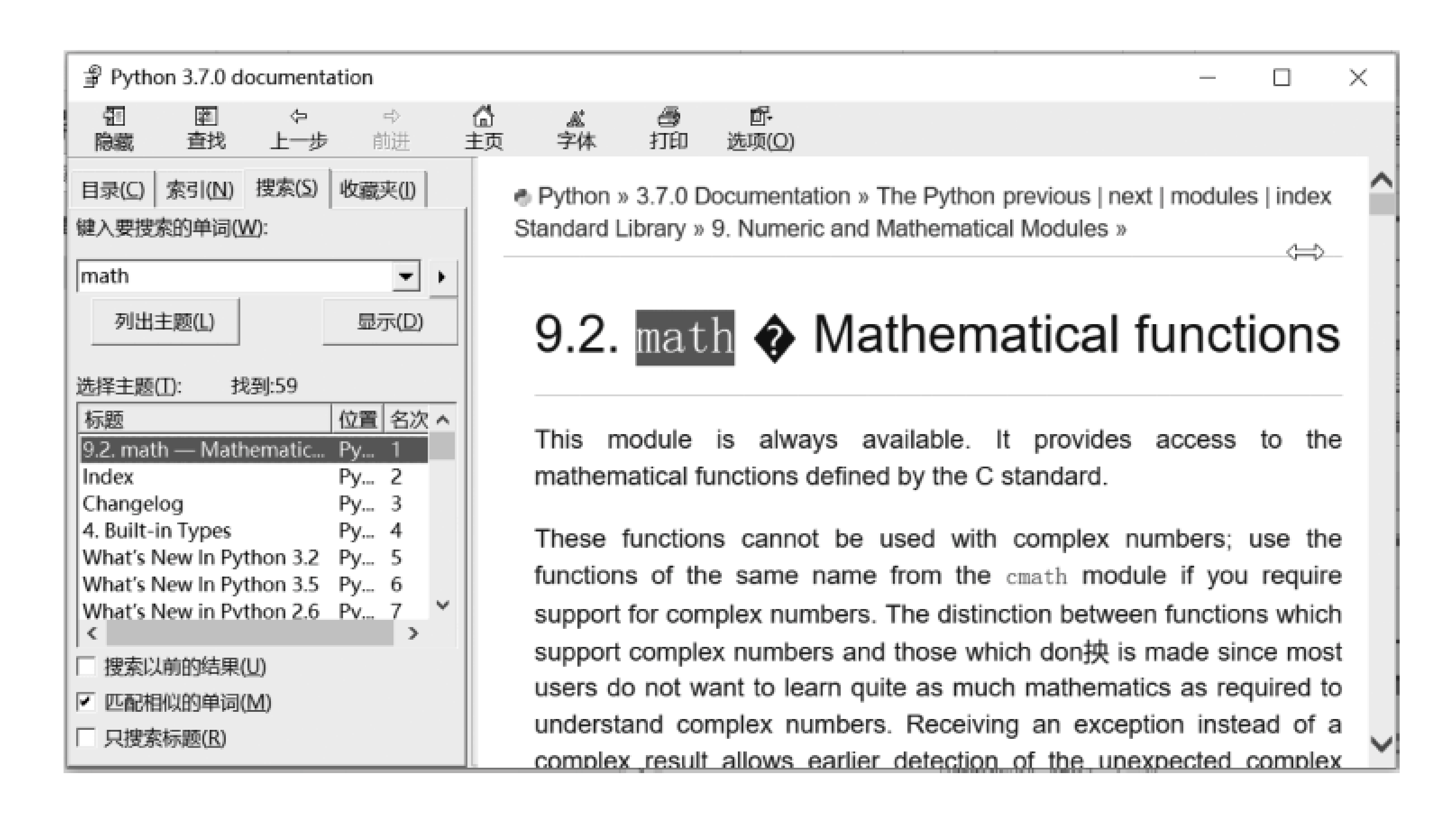


图 1-29　查找有关 math 模块的帮助信息

1.7.3　Python 官网

Python 官网的地址为 https://www.python.org/，如图 1-30 所示，用户可以从中下载各种版本的 Python 程序或者查看帮助文档等。

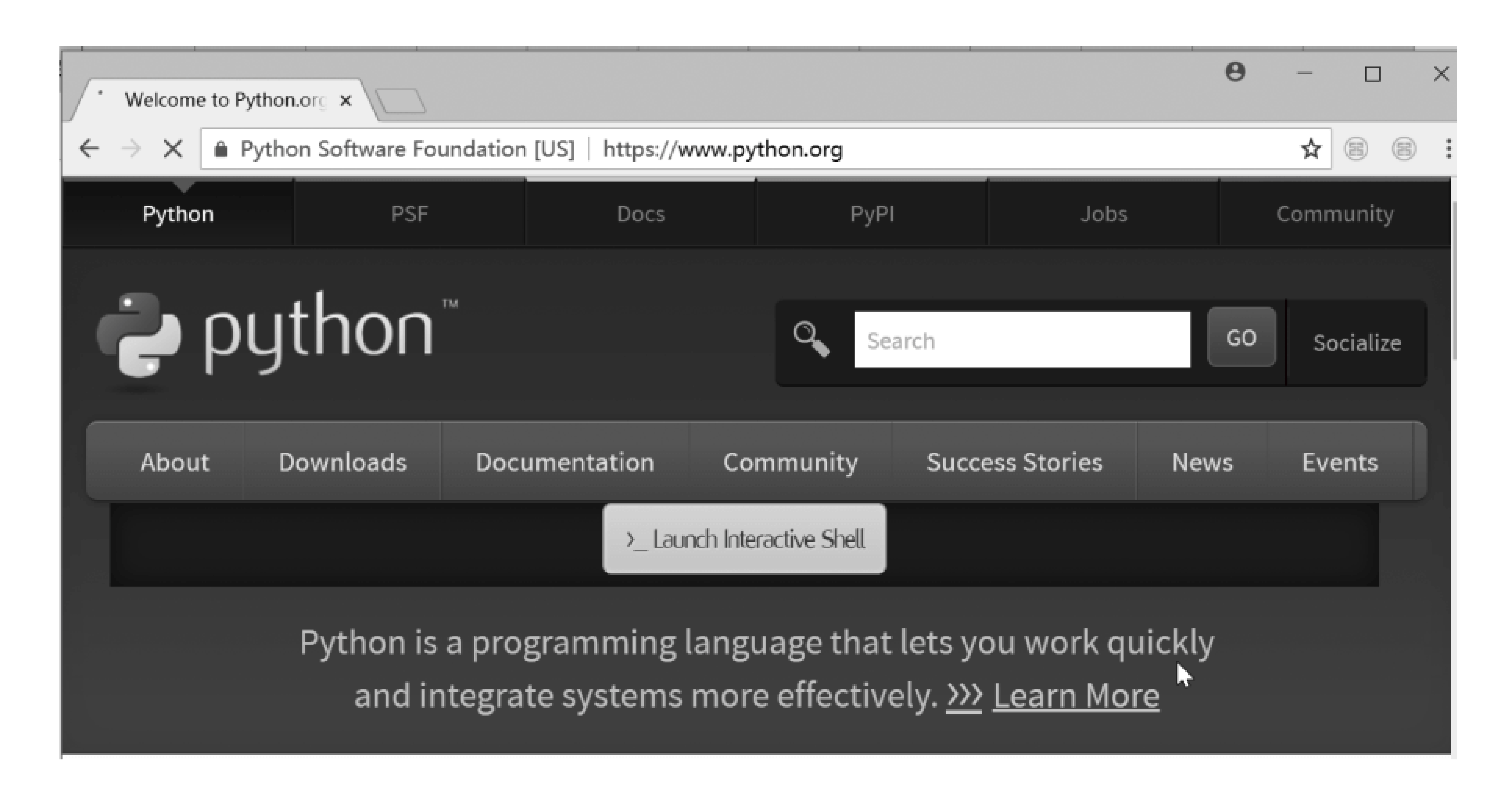


图 1-30　Python 官网

1.7.4　Python 扩展库索引

PyPI(Python Package Index)是 Python 官方的扩展库索引，所有人都可以下载第三方库或上传自己开发的库到 PyPI。PyPI 推荐使用 pip 包管理器来下载第三方库。

PyPI 官网的地址为 https://pypi.python.org/，如图 1-31 所示。

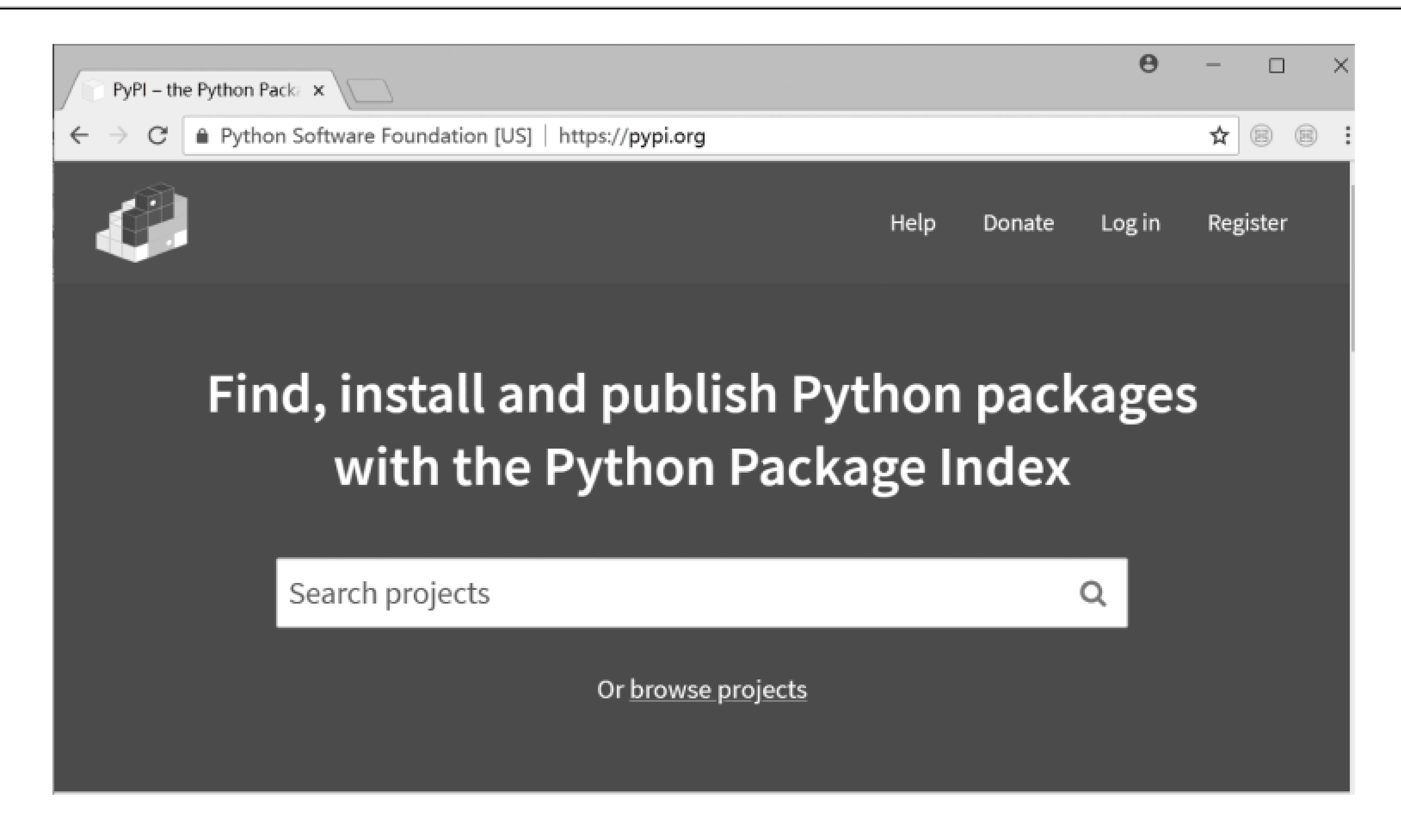


图 1-31 PyPI 官网

1.8 复 习 题

一、选择题

1. Python 语言属于________。
   A. 机器语言 B. 汇编语言 C. 高级语言 D. 以上都不是
2. 在下列选项中,不属于 Python 特点的是________。
   A. 面向对象 B. 运行效率高 C. 可移植性 D. 免费和开源
3. 在下列选项中,________是最常用的 Python 版本,也称之为 ClassicPython。
   A. CPython B. Jython C. IronPython D. PyPy
4. Python 内置的集成开发工具是________。
   A. PythonWin B. Pydev C. IDE D. IDLE
5. Python 解释器的提示符为________。
   A. > B. >> C. >>> D. #
6. 在 Python 解释器环境中,用于表示上一次运算结果的特殊变量为________。
   A. : B. _ C. > D. #
7. ________是 Python 官方的扩展库索引,所有人都可以下载第三方库或上传自己开发的库到其中。
   A. PyPI B. PyPy C. Pydev D. pip

二、填空题

1. Python 语言是一种解释型、面向________的计算机程序设计语言。
2. 用户编写的 Python 程序(避免使用依赖于系统的特性),无须修改就可以在任何支持 Python 的平台上运行,这是 Python 的________特性。
3. 在 Python 3.4 以后的版本中,________库用于安装管理 Python 扩展包,________库用于发布 Python 包。
4. 如果要关闭 Python 解释器,可以使用________命令或者按________组合键。
5. 在 Python 内置的集成开发环境 IDLE 中可以使用________键运行当前打开的源代码程序。
6. Python 注释以符号________开始,到行尾结束。

7. 在 Python 程序中导入 sys 模块后，可以通过列表________访问命令行参数。________表示 Python 脚本名；________表示第 1 个参数。

8. 在 Python 解释器中，使用________函数可以进入帮助系统；输入命令________可以退出帮助系统。

三、思考题

1. 简述 Python 语言的主要特点。
2. 简述 Python 语言的应用范围。
3. 简述 Python 2 和 Python 3 的主要区别。
4. Python 语言包括哪些实现？
5. Python 语言主要包括哪些集成开发环境？
6. 简述下载和安装 Python 的主要步骤。
7. 如何安装和管理 Python 扩展包？
8. 什么是 Python 解释器？如何使用 Python 解释器交互式测试 Python 代码？
9. Python 解释器环境中的特殊变量"_"表示什么含义？
10. 什么是 Python 源代码程序？如何运行 Python 源代码程序？
11. 如何使用文本编辑器和命令行编写和执行 Python 源代码程序？
12. 如何使用 Python 内置的集成开发环境 IDLE 编写和运行 Python 源代码程序？
13. 如何使用 Python 交互式帮助系统获取相关资源？
14. 如何使用 Python 文档获取 Python 语言及标准模块的详细参考信息？

1.9　上机实践

完成本章中的例 1.1～例 1.24，熟悉 Python 程序的编辑、开发和运行环境。

1.10　案例研究：安装和使用其他 Python 环境

Python 官网的安装程序默认不安装第三方库，而在实际项目工作中会使用大量的 Python 第三方库，因此可以直接安装包含大量常用库和 IDE 的 Python 环境，流行的有 Anaconda、Canopy、Python(x,y)、WinPython 等。

Anaconda 是 Python 的一个开源发行版本，主要面向科学计算。Anaconda 附带了 conda（包管理器）、Python 和 150 多个科学包及其依赖项。使用 Anaconda 无须花费大量时间安装众多的第三方 Python 包，可以立即开始处理数据。

在安装 Anaconda 后，相当于安装了 Python、IPython、集成开发环境 Spyder 以及一些常用的科学计算包。

本章案例研究的主要目的是通过 Anaconda 的安装和使用，帮助学生使用其他 Python 开发环境进行学习和研究。

限于篇幅，本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第2章

Python 语言基础



视频讲解

Python 程序由模块(即扩展名为.py 的源文件)组成。模块包含语句,语句是 Python 程序的基本构成元素。语句通常包含表达式,而表达式由操作数和运算符构成,用于创建和处理对象。Python 语言可以定义函数和类。本章简要概述 Python 语言的基础知识,后续章节将展开详细的阐述。

2.1 Python 程序概述

2.1.1 引例

【例 2.1】 已知三角形的 3 条边,求三角形的面积(area.py)。提示:假设 3 条边的边长分别为 a、b 和 c,则三角形的面积 $s=\sqrt{h*(h-a)*(h-b)*(h-c)}$,其中 h 为三角形周长的一半。

```
import math
a = 3.0
b = 4.0
c = 5.0
h = (a + b + c) / 2                          #三角形周长的一半
s = math.sqrt(h*(h-a)*(h-b)*(h-c))           #三角形的面积
print(s)
```

2.1.2 Python 程序的构成

Python 程序可以分解为模块、语句、表达式和对象。从概念上理解,其对应关系如下。

(1) Python 程序由模块组成,模块对应扩展名为.py 的源文件。一个 Python 程序由一个或多个模块构成。例 2.1 程序由模块 area.py 和内置模块 math 组成。

(2) 模块由语句组成。模块即 Python 源文件。在运行 Python 程序时按顺序依次执行模块中的语句。在例 2.1 程序中,import math 为导入模块语句;print(s)为调用函数表达式语句;其余的为赋值语句。

(3) 语句是 Python 程序的过程构造块,用于创建对象、变量赋值、调用函数、控制分支、创建循环、增加注释等。语句包含表达式。在例 2.1 程序中,语句 import math 用来导入 math 模块,并依次执行其中的语句;在语句“a=3.0”中,字面量 3.0 创建一个值为 3.0 的 float 型对象,并绑定到变量 a;在语句“h = (a + b + c)/2”中,算术表达式(a + b + c)/2 的运算结果为一个新的 float 型对象,并绑定到变量 h;“#”引导注释语句;在语句 print(s)中,调用内置函数 print(),输出对象 s 的值。

(4) 表达式用于创建和处理对象。在例 2.1 程序的语句“s = math.sqrt(h*(h-a)*

(h－b)＊(h－c))”中,表达式 h＊(h－a)＊(h－b)＊(h－c)的运算结果为一个新的 float 型对象,math.sqrt 调用模块 math 中的 sqrt()函数,计算参数对象的平方根。

2.2　Python 对象和引用

2.2.1　Python 对象概述

计算机程序通常用于处理各种类型的数据(即对象),不同的数据属于不同的数据类型,支持不同的运算操作。

在 Python 语言中,数据表示为对象。对象本质上是一个内存块,拥有特定的值,支持特定类型的运算操作。

在 Python 3 中,一切皆为对象。Python 语言中的每个对象由标识(identity)、类型(type)和值(value)标识。

(1) 标识用于唯一地标识一个对象,通常对应对象在计算机内存中的位置。使用内置函数 id(obj1)可以返回对象 obj1 的标识。

(2) 类型用于表示对象所属的数据类型(类),数据类型用于限定对象的取值范围以及允许执行的处理操作。使用内置函数 type(obj1)可以返回对象 obj1 所属的数据类型。

(3) 值用于表示对象的数据类型的值。使用内置函数 print(obj1)可以返回对象 obj1 的值。

通过内置的 type()函数可以判断一个对象的类型。通过内置的 id()函数可以获取一个对象唯一的 id 标识(CPython 的实现为内存存放位置)。

【例 2.2】 使用内置函数 type()、id()和 print()查看对象。

```
>>> 123                      #输出:123
>>> id(123)                  #输出:140706558370656
>>> type(123)                #输出:<class 'int'>
>>> print(123)               #输出:123
```

字面量 123 创建一个实例对象,其 id 标识为 140706558370656,类型为 int 类型,值为 123。

在 Python 3 中函数和类等也是对象,也具有相应的类型和 id。

【例 2.3】 查看 Python 的内置函数对象。

```
>>> type(abs)                #输出:<class 'builtin_function_or_method'>
>>> id(abs)                  #输出:2529313427104
>>> type(range)              #输出:<class 'type'>
>>> id(range)                #输出:140706557885440
```

2.2.2　使用字面量创建实例对象

对于内置对象,Python 通常提供使用字面量直接创建实例对象的语法。

Python 的数据类型定义了一个值的集合,在 Python 代码中使用字面量表示某个数据类型的值。例如,12、101 等表示 int 数据类型的值;0.17、3.14 等表示 float 数据类型的值;True 和 False 表示 bool 数据类型的值;'Hello, World'、'张三'等表示 str 数据类型的值。

字面量在 Python 语句中解释为表达式,Python 基于字面量创建相应的数据类型的对象。

【例 2.4】 使用字面量创建实例对象。

```
>>> 123                           #输出:123
>>> "abc"                         #输出:'abc'
```

Python 使用字面量 123 和"abc"分别创建一个 int 型对象和一个 str 型对象。

2.2.3 使用类对象创建实例对象

通过直接调用类对象可以创建实例对象,其语法格式如下。

类对象(参数)

【例 2.5】 使用类对象创建实例对象。

```
>>> int(12)                       #输出:12
>>> complex(1,2)                  #输出:(1+2j)
```

Python 使用 int(12)创建一个整数数据类型的实例对象;使用 complex(1,2)创建一个复数类型的实例对象。

另外,表达式的运算结果也可以创建新的对象;Python 语句 def 会创建函数对象;class 语句会创建类对象,详细阐述请参见本书的后续章节。

2.2.4 数据类型

在 Python 语言中,所有对象都有一个数据类型。Python 数据类型的定义为一个值的集合以及在这个值集上的一组运算操作。

例如整数数据类型(int),其值的集合为所有的整数,支持的运算操作包括+(加法)、-(减法)、*(乘法)、//(整除)等,88、1024 等都是整数类型数据。

每个对象存储一个值,例如,int 类型的对象可以存储值 1234、99 或 1333。不同的对象可以存储同一个值,例如,一个 str 类型的对象可以存储值'hello',另一个 str 类型的对象也可以存储值'hello'。在一个对象上可执行且只允许执行其对应数据类型定义的操作,例如,两个 int 对象可执行乘法运算,但两个 str 对象不允许执行乘法运算。

Python 数据类型包括内置数据类型和自定义数据类型。Python 语言提供了丰富的内置数据类型,用于有效地处理各种类型的数据。本书后续章节将展开内置数据类型和自定义数据类型的阐述。

2.2.5 变量和对象的引用

Python 对象是位于计算机内存中的一个内存数据块。为了引用对象,用户必须通过赋值语句把对象赋值给变量(也称之为把对象绑定到变量)。指向对象的引用即变量。

【例 2.6】 使用赋值语句把对象绑定到变量。

```
>>> a = 1                         #字面量表达式 1 创建值为 1 的 int 型实例对象,并绑定到变量 a
>>> b = 2                         #字面量表达式 2 创建值为 2 的 int 型实例对象,并绑定到变量 b
>>> c = a + b                     #表达式 a + b 创建值为 3 的 int 型实例对象,并绑定到变量 c
```

Python 使用字面量表达式 1、2 和表达式 a+b 创建 3 个整型对象,并使用赋值语句把 3 个对象分别绑定到变量 a、b 和 c。

字面量用于创建值为字面量的对象,即某个数据类型的实例对象;表达式使用运算符实现多个操作数(对象)的运算操作,并返回结果对象。用户可以把对象通过赋值语句赋值给一

个变量,即把对象绑定到一个变量,注意变量名必须为有效的标识符。

在 Python 3 中,作为对象的函数和类等也可以通过变量引用,但这样的引用一般意义不大,建议直接使用函数/类,以提高程序的可读性。例如:

```
>>> x = abs
>>> x(-123)                    #输出:123
>>> y = str
>>> id(y)                      #输出:140706557890640
>>> y.format('{0:.2f}',123)    #输出:'123.00'
```

2.2.6 Python 是动态类型语言

Python 是动态类型语言,即变量不需要显式声明数据类型。根据变量的赋值,Python 解释器自动确定其数据类型。

事实上,变量仅用于指向某个类型对象,因此变量可以不限定类型,即可以指向任何类型的对象。

通过标识符和赋值运算符(=)可以指定某个变量指向某个对象,即引用该对象。多个变量可以引用同一个对象,一个变量也可以改变指向其他的对象。

【例 2.7】 变量的动态类型示例。

```
>>> type(123)                  #输出:<class 'int'>
>>> id(123)                    #输出:140706558370656
>>> a = 123
>>> id(a)                      #输出:140706558370656
>>> b = 123
>>> id(b)                      #输出:140706558370656
>>> c = a
>>> id(c)                      #输出:140706558370656
>>> id('abc')                  #输出:2529314137232
>>> a = 'abc'
>>> id(a)                      #输出:2529314137232
```

123 为类 int 的对象实例,其 id 为 140706558370656; a=123,即变量 a 指向(引用)对象实例 123,故其 id 也为 140706558370656; b=123,即变量 b 也指向(引用)对象实例 123,故其 id 也为 140706558370656; c=a,变量 c 和变量 a 一样,指向(引用)对象实例 123,其 id 同样为 140706558370656。a='abc',变量 a 指向(引用)对象实例'abc',其 id 为 2529314137232。

2.2.7 Python 是强类型语言

Python 是一种强类型语言,每个变量指向的对象均属于某个数据类型,即只支持该类型允许的运算操作。

【例 2.8】 变量的强数据类型示例。

```
>>> a = 1                      #a 指向值为 1 的 int 型实例对象
>>> b = "11"                   #b 指向值为"11"的 str 型实例对象
>>> a + b                      #错误:int 型和 str 型对象不能直接相加,即 str 型对象不能自动转
                               #换为 int 型对象
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> b = 11                          #赋值语句:b 指向值为 11 的 int 型实例对象
>>> a + b                           #表达式运算结果,返回值为 12 的 int 型实例对象
```

2.2.8 对象内存示意图

Python 程序运行时会在内存中创建各种对象(位于堆内存中),通过赋值语句可以将对象绑定到变量(位于栈内存中),从而通过变量引用对象进行各种操作。

多个变量可以引用同一个对象。如果一个对象不再被任何有效作用域中的变量引用,则会通过自动垃圾回收机制回收该对象占用的内存。

为了更好地理解 Python 对象和变量的作用机制,本书采用对象内存示意图进行演示。

【例 2.9】 变量增量运算示例以及相应的对象内存示意图。

```
>>> i = 100
>>> i = i + 1
```

第 1 条语句,创建一个值为 100 的 int 对象,并绑定到变量 i;第 2 条语句,先计算表达式 i+1 的值,然后创建一个值为 101 的 int 对象,并绑定到变量 i。

执行各条语句后,其对象内存示意图如图 2-1 所示。

注意:在执行完第 2 条语句后,内存中存在 3 个 int 对象,即 100、1 和 101,变量 i 引用对象 101,其他两个对象没有被任何变量引用,将被自动垃圾回收器回收。

【例 2.10】 交换两个变量的示例以及相应的对象内存示意图。

```
>>> a = 123                         #a 指向值为 123 的 int 型实例对象
>>> b = 456                         #b 指向值为 456 的 int 型实例对象
>>> t = a                           #变量 t 和 a 一样,指向(引用)对象实例 123
>>> a = b                           #变量 a 和 b 一样,指向(引用)对象实例 456
>>> b = t                           #变量 b 和 t 一样,指向(引用)对象实例 123
```

在执行各条语句后,其对象内存示意图如图 2-2 所示。

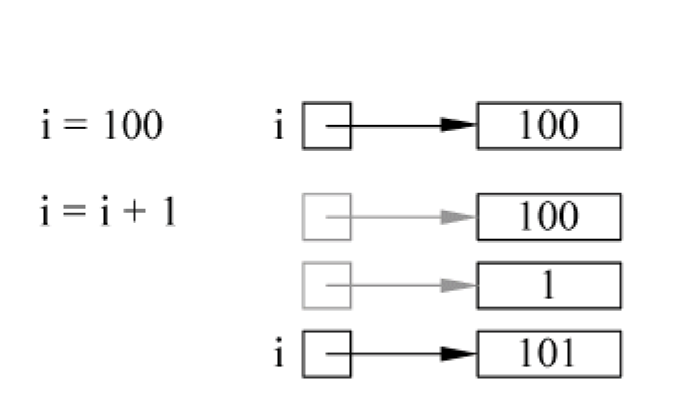


图 2-1 变量增量运算示例的对象内存示意图

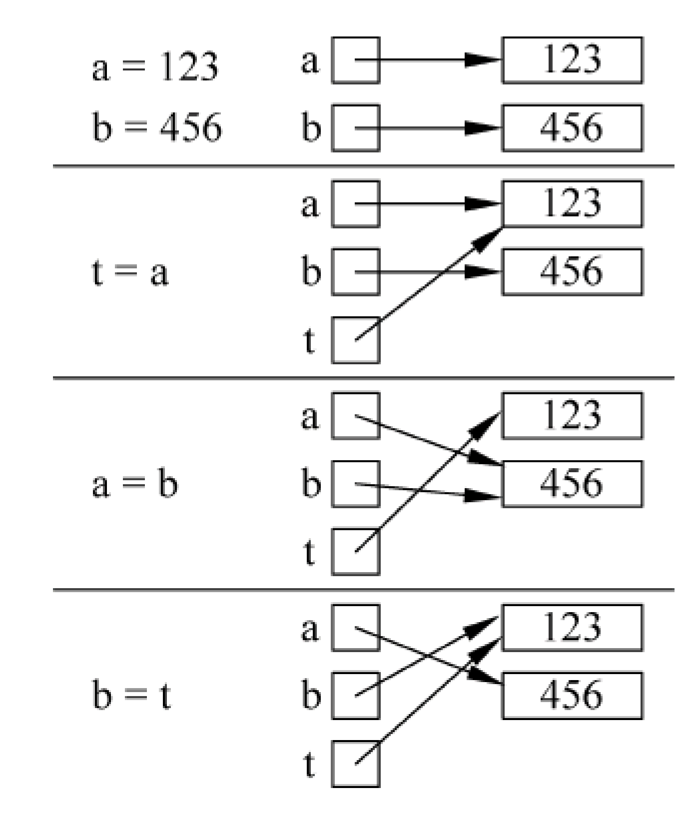


图 2-2 两个变量交换示例的对象内存示意图

2.2.9 对象的值比较和引用判别

通过==运算符可以判断两个变量指向的对象的值是否相同;通过 is 运算符可以判断两个变量是否指向同一对象。

【例 2.11】 对象的值比较(==)和引用判别(is)示例。

```
>>> x = 'abc'                  #x 指向值为"abc"的 str 型实例对象
>>> y = x                      #变量 y 和 x 一样,指向(引用)对象实例"abc"
>>> z = 'abcd'                 #z 指向值为"abcd"的 str 型实例对象
>>> x == y                     #输出:True
>>> x is y                     #输出:True
>>> x == z                     #输出:False
>>> x is z                     #输出:False
```

2.2.10 不可变对象和可变对象

Python 3 对象可以分为不可变对象(immutable)和可变对象(mutable)。不可变对象一旦创建,其值就不能被修改;可变对象的值可以被修改。Python 对象的可变性取决于其数据类型的设计,即是否允许改变其值。

Python 中的大部分对象都是不可变对象,例如 int、str、complex 等。变量是指向某个对象的引用,多个变量可以指向同一个对象。给变量重新赋值,并不改变原始对象的值,只是创建一个新对象,并指向该变量。

【例 2.12】 不可变对象示例。

```
>>> a = 18             #变量 a 指向 int 对象 18
>>> id(a)              #输出:140706365363776.表示 a 指向的 int 对象 18 的 id
>>> a = 25             #变量 a 指向 int 对象 25
>>> id(a)              #输出:140706365364000.表示 a 指向的 int 对象 25 的 id
>>> b = 25             #变量 b 指向 int 对象 25
>>> id(b)              #输出:140706365364000.表示 b 指向的 int 对象 25 的 id
>>> id(25)             #输出:140706365364000.表示 int 对象 25 的 id
```

对象本身的值可以改变的对象称为可变对象(例如 list、dict 等)。

【例 2.13】 可变对象示例。

```
>>> x = y = [1, 2, 3]  #变量 x 和 y 指向 list 对象[1, 2, 3]
>>> id(x)              #输出:1656936944328.表示变量 x 指向的 list 对象[1, 2, 3]的 id
>>> id(y)              #输出:1656936944328.表示变量 y 指向的 list 对象[1, 2, 3]的 id
>>> x.append(4)        #变量 x 指向的 list 对象[1, 2, 3]附加一个元素 4
>>> x                  #输出:[1, 2, 3, 4].表示变量 x 指向的 list 对象[1, 2, 3, 4]
>>> id(x)              #输出:1656936944328.变量 x 指向的 list 对象[1, 2, 3, 4]的 id 未改变
>>> x is y             #输出:True.表示变量 x 和 y 指向同一个 list 对象[1, 2, 3, 4]
>>> x == y             #输出:True.表示变量 x 和 y 指向的 list 对象值相等
>>> z = [1, 2, 3, 4]   #变量 z 指向的 list 对象[1, 2, 3, 4]
>>> id(z)              #输出:1656965757064.表示变量 z 指向的 list 对象[1, 2, 3, 4]的 id
>>> x is z             #输出:False.表示变量 x 和 z 指向不同的 list 对象[1, 2, 3, 4]
>>> x == z             #输出:True.表示变量 x 和 z 指向的 list 对象值相等
```

2.3 标识符及其命名规则

在 Python 语言中,包、模块、类、函数、变量等的名称必须为有效的标识符。

2.3.1 标识符

标识符是变量、函数、类、模块和其他对象的名称。标识符的第一个字符必须是字母、下画线("_"),其后的字符可以是字母、下画线或数字。一些特殊的名称,例如 if、for 等,作为

Python 语言的保留关键字,不能作为标识符。

例如,a_int、a_float、str1、_strname、func1 为正确的变量名;而 99var、It'sOK、for(关键字)为错误的变量名。

注意:

(1) Python 标识符区分大小写。例如,ABC 和 abc 视为不同的名称。

(2) 以双下画线开始和结束的名称通常具有特殊的含义。例如,__init__为类的构造函数,一般应避免使用。

(3) 避免使用 Python 预定义标识符名作为自定义标识符名。例如,NotImplemented、Ellipsis、int、float、list、str、tuple 等。

2.3.2 保留关键字

关键字即预定义保留标识符。关键字有特殊的语法含义,各关键字的使用将在后续章节陆续阐述。关键字不能在程序中用作标识符,否则会产生编译错误。Python 3 的关键字如表 2-1 所示。

表 2-1 Python 3 的关键字

关 键 字			
False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

【例 2.14】 使用 Python 帮助系统查看关键字。

(1) 运行 Python 内置集成开发环境 IDLE。

(2) 进入帮助系统。输入下列命令进入帮助系统:

```
>>> help()
```

(3) 查看 Python 关键字列表。输入下列命令查看 Python 关键字列表:

```
help> keywords
```

(4) 查看关键字 if 的帮助信息。输入下列命令查看 if 的帮助信息:

```
help> if
```

(5) 退出帮助系统。输入下列命令退出帮助系统:

```
help> quit
```

2.3.3 Python 预定义标识符

Python 语言中包含许多预定义内置类、异常、函数等,例如 float、ArithmeticError、print 等。用户应该避免使用 Python 预定义标识符名作为自定义标识符名。

使用 Python 的内置函数 dir(__builtins__)可以查看所有内置的异常名、函数名等。

使用“http://www.logilab.org/project/pylint”上提供的 pylint 工具可以检测 Python 源代码是否存在潜在的问题。

2.3.4 命名规则

Python 语言遵循的命名规则如表 2-2 所示。

表 2-2 Python 语言的命名规则

类型	命名规则	举例
模块/包名	全小写字母，简单、有意义，如果需要可以使用下画线	math、sys
函数名	全小写字母，可以使用下画线增加可阅读性	foo()、my_func()
变量名	全小写字母，可以使用下画线增加可阅读性	age、my_var
类名	采用 PascalCase 命名规则，即多个单词组成名称，每个单词除第一个字母大写外其余的字母均小写	MyClass
常量名	全大写字母，可以使用下画线增加可阅读性	LEFT、TAX_RATE

2.4 变量和赋值语句

计算机程序通常用于处理各种类型的数据(即对象)，不同的数据属于不同的数据类型，支持不同的运算操作。

计算机程序处理的数据必须放入内存。机器语言和汇编语言直接通过内存地址访问这些数据，而高级语言则通过内存单元命名(即变量)来访问这些数据。

在 Python 3 中一切皆为对象。对象是某个类(类型)的实例，对象由唯一的 id 标识。对象可以通过标识符来引用，对象引用即指向具体对象实例的标识符，也称之为“变量”。

2.4.1 变量的声明和赋值

变量的声明和赋值用于把一个变量绑定到某个对象，其语法格式如下。

```
变量名 = 字面量或表达式
```

最简单的表达式是字面量，Python 基于字面量的值创建一个对象，并绑定到变量；对于复杂的表达式，Python 先求值表达式，然后返回表达式结果对象，并绑定到变量。

Python 变量被访问之前必须初始化，即赋值(绑定到某个对象)，否则会报错。

【例 2.15】 变量的声明和赋值示例。

```
>>> x = 0; y = 0; z = 0        #变量 x、y 和 z 均指向 int 对象 0
>>> str1 = "abc"               #变量 str1 指向值为"abc"的 str 型实例对象
>>> aFloat                     #变量 aFloat 未声明和定义(NameError: name 'aFloat' is not defined)
```

2.4.2 链式赋值语句

链式赋值(chained assignment)的语句形式如下：

```
变量 1 = 变量 2 = 表达式
```

等价于：

```
变量 2 = 表达式
变量 1 = 变量 2
```

链式赋值用于为多个变量赋同一个值。

【例 2.16】 链式赋值语句示例。

```
>>> x = y = 123              # 变量 x 和 y 均指向 int 对象 123
>>> x                        # 输出:123
>>> y                        # 输出:123
```

2.4.3 复合赋值语句

复合赋值运算符不仅可以简化程序代码,使程序精练,而且可以提高程序的效率。Python 中的复合赋值运算符如表 2-3 所示。

表 2-3 复合赋值运算符

运 算 符	含 义	举 例	等 效 于
+=	加法赋值	sum += item	sum = sum + item
	字符串拼接	aStr +="Foo"	aStr = aStr +"Foo"
-=	减法赋值	count -=1	count = count - 1
*=	乘法赋值	x *= y+5	x = x * (y+5)
/=	除法赋值	x /= y-z	x = x / (y-z)
//=	整除赋值	x //= y-z	x = x // (y-z)
%=	取模赋值	x %= 2	x = x % 2
**=	幂运算赋值	x **= 2	x = x ** 2
<<=	左移赋值	x <<= y	x = x << y
>>=	右移赋值	x >>= y	x = x >> y
&=	按位与赋值	x &=y	x = x &y
|=	按位或赋值	x |=y	x = x |y
^=	按位异或赋值	x ^= y	x = x ^ y

【例 2.17】 复合赋值示例。

```
>>> i = 1         # 变量 i 指向 int 对象 1
>>> i += 1        # 先计算表达式 i + 1 的值,然后创建一个值为 2 的 int 对象,并绑定到变量 i
>>> i             # 输出:2
>>> i * = 3       # 先计算表达式 i * 3 的值,然后创建一个值为 6 的 int 对象,并绑定到变量 i
>>> i             # 输出:6
```

2.4.4 删除变量

用户可以使用 del 语句删除不再使用的变量。

【例 2.18】 删除变量示例。

```
>>> x = 1         # 变量 x 指向 int 对象 1
>>> del x         # 删除变量 x
>>> x             # 变量 x 未声明和定义(NameError: name 'x' is not defined)
```

2.4.5 序列解包赋值

Python 支持将序列数据类型(参见第 5 章)解包为对应相同个数的变量。

【例 2.19】 序列解包示例。

```
>>> a,b = 1,2     # 变量 a 指向 int 对象 1,变量 b 指向 int 对象 2
```

```
>>> a                     # 输出:1
>>> b                     # 输出:2
```

注意：变量的个数必须与序列的元素个数一致，否则会产生错误。例如，对于语句"x,y=(1,2,3)"，由于右侧的元组序列包含 3 个元素，但是左侧只有两个变量，所以会产生错误。

如果只需要解包部分值，则可以采用特殊变量"_"。例如：

```
>>> _, share, price, _ = [ 'ACME', 50, 102.11, (2018, 8, 21)]
>>> share                           # 输出:50
>>> price                           # 输出:102.11
```

【例 2.20】 使用序列解包实现变量交换。

```
>>> a,b = (1,2)                     # 变量 a 指向 int 对象 1,变量 b 指向 int 对象 2
>>> a,b = b,a                       # 变量 a 和 b 的值进行交换
>>> a                               # 输出:2
>>> b                               # 输出:1
```

说明：在 Python 语言中，使用"a,b=b,a"的语句方式可以"优雅地"实现两个变量的值的交换。

2.4.6 常量

Python 语言不支持常量，即没有语法规则限制改变一个常量的值。Python 语言使用约定，声明在程序运行过程中不会改变的变量为常量，通常使用全大写字母（可以使用下画线增加可阅读性）表示常量名。

【例 2.21】 常量示例。

```
>>> TAX_RATE = 0.17                 # 浮点类型常量 TAX_RATE
>>> PI = 3.14                       # 浮点类型常量 PI
>>> ECNU = '华东师范大学'           # 字符串常量 ECNU
```

2.5 表达式和运算符

2.5.1 表达式的组成

表达式是可以计算的代码片段，由操作数和运算符构成。操作数、运算符和圆括号按一定的规则组成表达式。表达式通过运算后产生运算结果，返回结果对象。运算结果对象的类型由操作数和运算符共同决定。

运算符表明对操作数进行什么样的运算。运算符包括+、-、*、/等。操作数包括文本常量（没有名称的常数值，例如 1、"abc"）、变量（例如 i=123）、类的成员变量/函数（例如 math.pi、math.sin(x)）等，也可以包含子表达式（例如(2 ** 10)）。

表达式既可以非常简单，也可以非常复杂。当表达式包含多个运算符时，运算符的优先级控制各个运算符的计算顺序。例如，表达式 x+y*z 按 x+(y*z)计算，因为 * 运算符的优先级高于+运算符。

【例 2.22】 表达式示例。

```
>>> import math                     # 导入 math 模块
>>> a = 2;b = 10                    # 变量 a 指向 int 对象 2,变量 b 指向 int 对象 10
>>> a + b                           # 输出:12
```

```
>>> math.pi                  #输出:3.141592653589793
>>> math.sin(math.pi/2)      #输出:1.0
```

2.5.2 表达式的书写规则

Python 表达式遵循下列书写规则。

(1) 表达式从左到右在同一个基准上书写。例如,数学公式 a^2+b^2 应该写为 a ** 2+b ** 2。

(2) 乘号不能省略。例如,数学公式 ab(表示 a 乘以 b)应写为 a * b。

(3) 括号必须成对出现,而且只能使用圆括号;圆括号可以嵌套使用。

【例 2.23】 复杂表达式示例。

数学表达式 $\frac{1}{2}\sin[a(x+1)+b]$ 写成 Python 表达式为 math.sin(a * (x+1)+b)/2。

2.5.3 运算符概述

Python 运算符用于在表达式中对一个或多个操作数进行计算并返回结果值,接受一个操作数的运算符被称作一元运算符,例如正负号运算符+或−;接受两个操作数的运算符被称作二元运算符,例如算术运算符+、−、*、/等。

如果一个表达式中包含多个运算符,则计算顺序取决于运算符的结合顺序和优先级。

优先级高的运算符优先计算,例如,在 1+2*3 中*的优先级比+高,故先计算 2*3。同一优先级的运算符按结合顺序依次计算,例如+、−(以及*、/)为同一优先级左结合的运算符,故 1+2−3 等同于(1+2)−3;2*4/2 等同于(2*4)/2。注意,赋值运算符=为右结合运算符,故 a=b=c 等同于 a=(b=c)。用户可以使用圆括号“()”强制改变运算顺序。

【例 2.24】 表达式中运算符的优先级示例。

```
>>> 11 + 22 * 3              #输出:77
>>> (11 + 22) * 3            #输出:99
```

2.5.4 Python 运算符

Python 语言定义了许多运算符,按优先顺序排列如表 2-4 所示。本书后续章节将陆续阐述。通过运算符重载(overload)可以为用户自定义的类型定义新的运算符。

表 2-4 Python 运算符

运 算 符	描 述
lambda	Lambda 表达式
or	布尔“或”
and	布尔“与”
not x	布尔“非”
in、not in	成员测试
is、is not	同一性测试
<、<=、>、>=、!=、==	比较
|	按位或
^	按位异或
&	按位与
<<、>>	移位

续表

运　算　符	描　　述
+、-	加法与减法
*、/、%、//	乘法、除法、取余、整数除法
+x、-x	正负号
~x	按位翻转
**	指数/幂
x.attribute	属性参考
x[index]	索引访问
x[index:index]	切片操作
f(arguments…)	函数调用
(expression,…)	绑定或元组显示
[expression,…]	列表显示
{key:datum,…}	字典显示
'expression,…'	字符串转换

2.6 语　　句

2.6.1 Python 语句

语句是 Python 程序的过程构造块，用于定义函数、定义类、创建对象、变量赋值、调用函数、控制分支、创建循环等。

Python 语句分为简单语句和复合语句。

简单语句包括表达式语句、赋值语句、assert 语句、pass 语句、del 语句、return 语句、yield 语句、raise 语句、break 语句、continue 语句、import 语句、global 语句、nonlocal 语句等。

复合语句包括 if 语句、while 语句、for 语句、try 语句、with 语句、函数定义、类定义等。

Python 语句涉及许多程序构造要素，将在本书后续章节陆续阐述。

【例 2.25】 Python 语句示例(statement.py)：输入圆的半径 r，计算并输出圆的周长和面积。

```
import math                          #import 语句,用于导入 math 模块
r = float(input("请输入圆的半径 r:"))  #赋值语句.输入圆的半径 r,并转换为 float 数据类型
p = 2 * math.pi * r                  #赋值语句.计算圆的周长
s = math.pi * r * * 2                #赋值语句.计算圆的面积
print("圆的周长为:", p)               #表达式语句.输出圆的周长
print("圆的面积为:", s)               #表达式语句.输出圆的面积
```

程序运行结果如下。

```
请输入圆的半径 r: 5
圆的周长为: 31.41592653589793
圆的面积为: 78.53981633974483
```

2.6.2 Python 语句的书写规则

Python 语句的书写规则如下。

(1) 使用换行符分隔，在一般情况下一行一条语句。

（2）从第 1 列开始，前面不能有任何空格，否则会产生语法错误。注意，注释语句可以从任意位置开始；复合语句构造体必须缩进。例如：

```
>>>                                   #正确
>>>    print("abc")                   #报错.IndentationError: unexpected indent
```

（3）反斜杠（\）用于一个代码跨越多行的情况。如果语句太长，可以使用续行符（\）。三引号定义的字符串（"""…"""）、元组（(…)）、列表（[…]）、字典（{…}）可以放在多行，而不必使用续行符（\），因为它们可以清晰地表示定义的开始和结束。例如：

```
>>> print("如果语句太长,可以使用续行符(\),\
续行内容.")
```

（4）分号（;）用于在一行书写多条语句。例如：

```
>>> a = 0; b = 0; c = 0              #变量 a、b 和 c 均指向 int 对象 0
>>> s = "abc";print(s)               #变量 s 指向值为"abc"的 str 型实例对象,并输出 abc
```

2.6.3 复合语句及其缩进书写规则

由多行代码组成的语句称为复合语句。复合语句（条件语句、循环语句、函数定义和类定义，例如 if、for、while、def、class 等）由头部语句（header line）和构造体语句块（suites）组成。构造体语句块由一条或多条语句组成。复合语句和构造体语句块的缩进书写规则如下。

（1）头部语句由相应的关键字（例如 for）开始，构造体语句块则为下一行开始的一行或多行缩进代码。例如：

```
>>> sum = 0
>>> for i in range(1,11):
        sum = sum + i
        print(i, end = ' ')
1 2 3 4 5 6 7 8 9 10                 #输出:1 2 3 4 5 6 7 8 9 10
>>> print(sum)                       #输出:55
```

（2）通常缩进是相对头部语句缩进 4 个空格，也可以是任意空格，但同一构造体代码块的多条语句缩进的空格数必须一致。如果语句不缩进，或缩进不一致，将导致编译错误。注意，Python 强制缩进，以保证源代码的规范性和可读性。另外，Python 不建议使用制表符缩进，因为制表符在不同系统中产生的缩进效果可能不一致。

（3）如果条件语句、循环语句、函数定义和类定义比较短，可以放在同一行。例如：

```
>>> for i in range(1,11): print(i, end = ' ')
```

2.6.4 注释语句

Python 注释语句以符号“#”开始，到行末结束。Python 注释语句可以出现在任何位置。Python 解释器将忽略所有的注释语句，注释语句不会影响程序的执行结果。良好的注释可以帮助用户阅读和理解程序。

【例 2.26】 注释语句示例。

```
>>> #A "hello world!" program
>>>                                   #注释可以在任意位置,以#开始,到行末结束
>>> print("hello world")              # 输出:hello world
```

Python 模块、类和函数可以定义规范的注释信息，以生成帮助文档，相关内容将在后续章节阐述。

2.6.5 空语句

如果要表示一个空的代码块,可以使用 pass 语句。

【例 2.27】 空语句示例。

```
>>> def do_nothing():
        pass
```

2.7 函数和模块

Python 语言中包括许多内置的函数,例如 print()、max()等,用户也可以自定义函数。函数是可以重复调用的代码块,使用函数可以有效地组织代码,提高代码的重用率。

本节简要介绍函数的定义和调用,有关函数的展开阐述请参见第 8 章。

2.7.1 函数的创建和调用

Python 使用复合语句 def 创建函数对象,其语法格式如下。

```
def 函数名([形参列表]):
    函数体
```

函数的调用格式如下。

```
函数名([实参列表])
```

在创建函数时可以声明函数的参数,即形式参数,简称形参;在调用函数时需要提供函数需要的参数的值,即实际参数,简称实参。

函数可以使用 return 返回值。无返回值的函数相当于其他编程语言中的过程。

【例 2.28】 声明和调用函数示例(sayHello.py)。

```
def sayHello():                                        #创建函数对象 sayHello
    print('Hello World!')                              #函数体
    print('To be or not to be, this is a question!')   #函数体
sayHello()                                             #调用函数 sayHello()
```

程序运行结果如下。

```
Hello World!
To be or not to be, this is a question!
```

【例 2.29】 声明和调用函数 getValue(b, r, n),根据本金 b、年利率 r 和年数 n 计算最终收益 v。提示:$v=b(1+r)^n$。

```
def getValue(b,r,n):            #创建函数对象 getValue
    v = b*((1+r)**n)            #计算最终收益 v
    return v                    #使用 return 返回值
total = getValue(1000,0.05,5)   #调用函数 getValue()
print(total)                    #打印结果
```

程序运行结果如下。

```
1276.2815625000003
```

2.7.2 内置函数

Python 语言中包含若干常用的内置函数，例如 dir()、type()、id()、help()、len()等，用户可以直接使用。

【例 2.30】 内置函数使用示例。

```
>>> s = "To be or not to be, this is a question!"
>>> type(s)                              #返回对象 s 所属的数据类型.输出:<class 'str'>
>>> len(s)                               #返回字符串 s 的长度.输出:39
```

2.7.3 模块函数

通过 import 语句可以导入模块 module，然后使用 module.function(arguments)形式调用模块中的函数。

【例 2.31】 模块的导入示例 1。

```
>>> import math
>>> math.sin(2)                          #输出:0.9092974268256817
```

用户也可以通过"from…import…"形式直接导入包中的常量、函数和类，或者通过"from…import *"形式导入包中的所有元素，然后使用 function(arguments)形式直接调用模块中的函数。

【例 2.32】 模块的导入示例 2。

```
>>> from math import sin
>>> sin(2)                               #输出:0.9092974268256817
```

2.7.4 函数 API

Python 语言中提供了海量的内置函数、标准库函数、第三方模块函数，使用这些函数的关键是了解其调用方法，函数的调用方法由应用程序编程接口（API）确定。常用函数 API 如表 2-5 所示。

表 2-5 Python 常用函数 API

模块	函数调用方法（签名）	功能描述
内置函数	print(x)	输出 x
	abs(x)	x 的绝对值
	type(o)	o 的类型
	len(a)	a 的长度
Python 标准库 math 模块中的函数	math.sin(x)	x 的正弦（参数以弧度为单位）
	math.cos(x)	x 的余弦（参数以弧度为单位）
	math.exp(x)	x 的指数函数（即 e^x）
	math.log(x, b)	x 的以 b 为底的对数（即 $\log_b x$）。底数为 e，即自然对数（即 $\log_e x$）
	math.sqrt(x)	x 的平方根
Python 标准库 random 模块中的函数	random.random()	返回[0,1)数据区间的随机浮点数
	random.randrange(x, y)	返回[x,y)数据区间的随机整数，其中 x 和 y 均为整数

Python 典型的函数调用如表 2-6 所示。

表 2-6 Python 典型的函数调用

函数调用	返回值	说明
print('Hello')	在控制台输出字符串 Hello	内置函数
len('Hello')	5	内置函数
math.sin(1)	0.8414709848078965	math 模块中的函数
math.sqrt(-1.0)	运行时错误	负数的平方根
random.random()	0.3151503393010261	random 模块中的函数。注意，每次产生不同的随机数

2.8 类和对象

类和对象是面向对象编程的两个主要方面，有关面向对象的展开阐述请参见第 9 章。

2.8.1 创建类对象

Python 使用复合语句 class 创建类对象，其语法格式如下。

```
class 类名:
    类体
```

在类体中可以定义属于类的属性、方法等。

2.8.2 实例对象的创建和调用

基于类对象可以创建其实例对象，然后访问其方法或属性。其语法格式如下：

```
anObject = 类名(参数列表)
anObject.对象方法
```

或

```
anObject.对象属性
```

【例 2.33】 类和对象示例(Person.py)：定义类 Person，创建其对象，并调用对象方法。

```
class Person:                           #定义类 Person
    def sayHello(self):                 #定义类 Person 的函数 sayHello()
        print('Hello, how are you?')
p = Person()                            #创建对象
p.sayHello()                            #调用对象的方法
```

程序运行结果如下。

```
Hello, how are you?
```

2.9 模块和包

在 Python 语言中，包含 Python 代码的源文件(通常包含用户自定义的变量、函数和类)称为模块，其扩展名为.py。功能相近的模块可以组织成包，包是模块的层次性组织结构。有关模块和包的展开阐述请参见第 10 章。

在 Python 标准库和第三方库中提供了大量的模块，通过 import 语句可以导入模块，并使用其定义的功能。导入和使用模块功能的基本形式如下。

```
import 模块名                          #导入模块
模块名.函数名                          #使用包含模块的全限定名称调用模块中的函数
模块名.变量名                          #使用包含模块的全限定名称访问模块中的变量
```

【例 2.34】 模块和包示例(module1.py)：求解一元二次方程 $x^2+5x+6=0$。

```
import math                                  #导入标准模块 math
a = 1; b = 5; c = 6                          #变量 a、b 和 c 分别指向 int 对象 1、5 和 6
x1 = (-b + math.sqrt(b*b - 4*a*c))/(2*a)     #使用模块 math 中的函数 sqrt()求解平方根
x2 = (-b - math.sqrt(b*b - 4*a*c))/(2*a)
print('方程 x*x + 5*x + 6 = 0 的解为:', x1, x2) #输出一元二次方程的两个解
```

程序运行结果如下。

```
方程 x*x + 5*x + 6 = 0 的解为: -2.0 -3.0
```

2.10 复 习 题

一、选择题

1. 在 Python 中，以下标识符合法的是________。
   A. _　　B. 3C　　C. it's　　D. str
2. 在 Python 表达式中可以使用________控制运算的优先顺序。
   A. 圆括号()　　B. 方括号[]　　C. 花括号{}　　D. 尖括号<>
3. 在下列 Python 语句中非法的是________。
   A. x=y = 1　　B. x = (y = 1)　　C. x, y = y, x　　D. x = 1; y=1
4. 以下 Python 注释代码不正确的是________。
   A. # Python 注释代码
   B. #Python 注释代码 1 #Python 注释代码 2
   C. """ Python 文档注释"""
   D. // Python 注释代码
5. 数学关系式 2<x≤10 表示成正确的 Python 表达式为________。
   A. 2<x<=10　　B. 2<x and x<=10
   C. 2<x && x <=10　　D. x>2 or x<=10
6. 在 Python 中，以下赋值语句正确的是________。
   A. x+y=10　　B. x=2y　　C. x=y=30　　D. 3y=x+1
7. 为了给整型变量 x、y、z 赋初值 10，下面 Python 赋值语句正确的是________。
   A. xyz=10　　B. x=10 y=10 z=10
   C. x=y=z=10　　D. x=10,y=10,z=10
8. 为了给整型变量 x、y、z 赋初值 5，下面 Python 赋值语句正确的是________。
   A. x=5; y=5; z=5　　B. xyz=5
   C. x,y,z=5　　D. x=5,y=5,z=5
9. 已知 x=2 并且 y=3，复合赋值语句 x *= y + 5 执行后 x 变量中的值是________。
   A. 11　　B. 16　　C. 13　　D. 26
10. 在整型变量 x 中存放了一个两位数，如果要将该两位数的个位数字和十位数字交换

位置，例如将 13 变成 31，以下 Python 表达式正确的是________。

A. (x % 10) * 10 + x // 10　　B. (x % 10) // 10 + x // 10
C. (x / 10) % 10 + x // 10　　D. (x % 10) * 10 + x % 10

11. 下列与数学表达式 $\frac{cd}{2ab}$ 对应的 Python 表达式不正确的是________。

A. c*d/(2*a*b)　B. c/2*d/a/b　C. c*d/2*a*b　D. c*d/2/a/b

二、填空题

1. Python 语句分为________语句和复合语句。
2. Python 使用________格式划分语句块。
3. 在 Python 中如果语句太长，可以使用________作为续行符。
4. 在 Python 中一行书写两条语句时，语句之间可以使用________作为分隔符。
5. Python 使用符号________标示注释。
6. 在 Python 中要表示一个空的代码块，可以使用空语句________。
7. 计算 $2^{32}-1$ 的 Python 表达式可以书写为________。
8. Python 表达式 4.5/2、4.5//2 和 4.5%2 的值分别为________。
9. Python 表达式 12/4−2+5*8/4%5/2 的值为________。
10. Python 中的大部分对象均为不可变对象，例如________等，________等则为可变对象。
11. Python 提供了两个对象身份比较运算符________和________来测试两个变量是否指向同一个对象；通过内置函数________来测试对象的类型；通过________运算符判断两个变量指向的对象的值是否相同。
12. Python 语句序列“a,b=3,4; a,b = b,a; print(a,b)”的执行结果是________。

三、思考题

1. Python 语句的主要作用是什么？Python 中主要包含哪些语句？
2. Python 中 pass 语句的作用是什么？
3. Python 中 type(1)的含义是什么？
4. 在 Python 中有哪几种注释方式？
5. Python 语句的主要书写规则是什么？
6. Python 表达式遵循哪些主要的书写规则？
7. 假设有 a=10，写出下面表达式运算后 a 的值。

(1) a += a　　(2) a -= 2　　(3) a *= 2 + 3
(4) a /= 2 + 3　　(5) a %= a - a % 4　　(6) a //= a - 3

8. 当运行测试输入 6789 时，写出下面 Python 程序的执行结果。

```
num = int(input("请输入一个整数:"))
while (num != 0):
        print(num % 10,end='')
        num = num // 10
```

9. 下列 Python 语句的输出结果是________。

```
def f(): pass
print(type(f()))
```

10. 下列 Python 语句的输出结果是________。

```
x = y = [1, 2]; x.append(3)
print(x is y, x == y,end='')
```

```
z = [1, 2, 3]
print(x is z,x == z,y == z)
```

2.11 上机实践

1. 完成本章中的例 2.1～例 2.34,熟悉 Python 语言基础知识的应用实践。

2. 编写程序,输入本金、年利率和年数,计算复利(结果保留两位小数),运行效果参见图 2-3。

提示:

用户可以使用“print(str.format("本金利率和为:{0:2.2f}",amount))”的语句形式输出程序运行效果(结果保留两位小数)。

3. 编写程序,输入球的半径,计算球的表面积和体积(结果保留两位小数),运行效果参见图 2-4。

```
请输入本金:2000
请输入年利率:5.6
请输入年数:5
本金利率和为:2626.33
```

图 2-3 计算复利的运行效果

```
请输入球的半径:2.5
球的表面积为:78.54,体积为:65.45
```

图 2-4 计算球的表面积和体积的运行效果

提示:

(1) 球的表面积的计算公式为 $4\pi r^2$,球的体积的计算公式为 $\frac{4}{3}\pi r^3$。

(2) 用户可以使用“print(str.format("球的表面积为:{0:2.2f},体积为:{1:2.2f}",area,volume))”的语句形式输出程序运行效果。

4. 编写程序,声明函数 getValue(b,r,n),根据本金 b、年利率 r 和年数 n 计算最终收益 v,$v = b(1 + r)^n$;然后编写测试代码,提示输入本金、年利率和年数,显示最终收益(保留两位小数)。

5. 编写程序,求解一元二次方程 $x^2 - 10x + 16 = 0$,运行效果参见图 2-5。

6. 编写程序,提示输入姓名和出生年份,输出姓名和年龄,运行效果参见图 2-6。

```
方程x*x - 10*x + 16 = 0的解为:8.0 2.0
```

图 2-5 求解一元二次方程的运行效果

```
请输入您的姓名:Mary
请输入您的出生年份:2001
您好!Mary。您17岁。
```

图 2-6 输出姓名和年龄的运行效果

提示:

(1) 用户可以使用 datetime.date.today().year 返回当年的年份值。

(2) 用户可以使用“print("您好!{0}。您{1}岁。".format(sName,age))”的语句形式输出程序运行效果。

2.12 案例研究:使用 Pillow 库处理图像文件

本章讨论了 Python 模块、对象、方法和函数等基本知识。本章案例研究使用 Python 图像处理库 Pillow 中的模块和对象来处理图像,实现读取图像、获取图像信息、调整图像大小、旋转图像、平滑图像、剪切图像等基本图像处理任务。

本章案例研究的主要目的是在具体展开 Python 语言细节之前，帮助学生了解使用由第三方开发的开源 Python 软件库来解决实际问题的基本思路和方法。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 3 章

程序流程控制



视频讲解

在 Python 程序中，对于语句的执行有 3 种基本控制结构，即顺序结构、选择结构、循环结构。

3.1 顺序结构

若程序中的语句按各语句出现位置的先后次序执行，称之为顺序结构，参见图 3-1。在图 3-1 中先执行语句块 1，再执行语句块 2，最后执行语句块 3，3 个语句块之间是顺序执行关系。

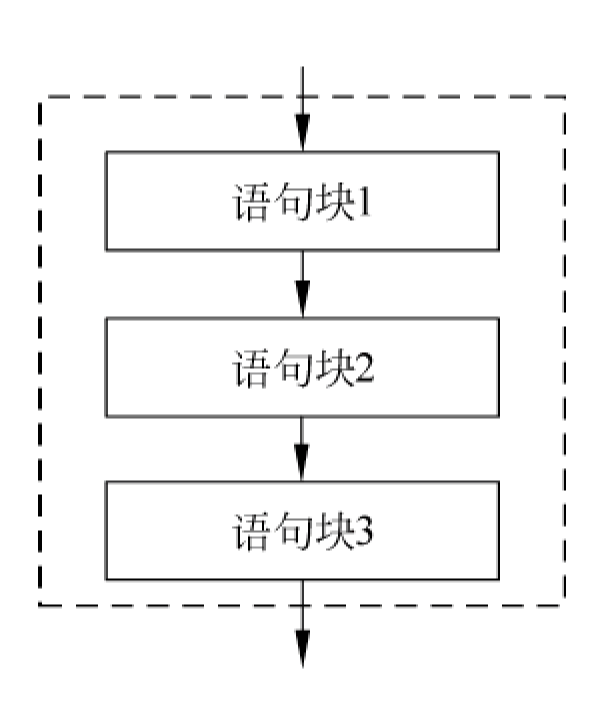


图 3-1 顺序结构示意图

【例 3.1】 顺序结构示例(area. py)：输入三角形 3 条边的边长(为简单起见，假设这 3 条边可以构成三角形)，计算三角形的面积。提示：三角形面积$=\sqrt{h*(h-a)*(h-b)*(h-c)}$，其中，a、b、c 是三角形 3 条边的边长，h 是三角形周长的一半。

```
import math
a = float(input("请输入三角形的边长 a:"))
b = float(input("请输入三角形的边长 b:"))
c = float(input("请输入三角形的边长 c:"))
h = (a + b + c) / 2                          #三角形周长的一半
area = math.sqrt(h*(h-a)*(h-b)*(h-c));       #三角形面积
print(str.format("三角形三边分别为:a={0},b={1},c={2}", a, b, c))
print(str.format("三角形的面积 = {0}", area))
```

程序运行结果如下。

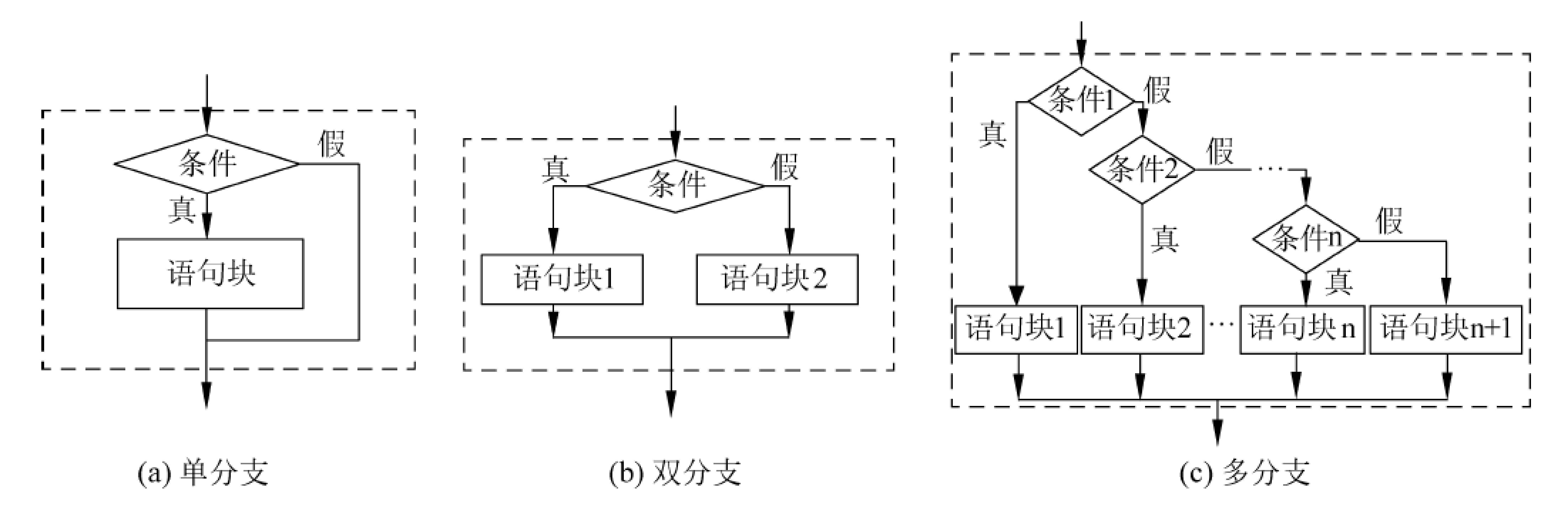
```
请输入三角形的边长 a: 3
请输入三角形的边长 b: 4
请输入三角形的边长 c: 5
三角形三边分别为: a=3.0,b=4.0,c=5.0
三角形的面积 = 6.0
```

3.2 选择结构

选择结构可以根据条件来控制代码的执行分支，也叫分支结构。Python 使用 if 语句来实现分支结构。

3.2.1 分支结构的形式

分支结构包含单分支、双分支和多分支等形式，流程如图 3-2(a)～(c)所示。



(a) 单分支　(b) 双分支　(c) 多分支

图 3-2 if 语句的选择结构

3.2.2 单分支结构

if 语句单分支结构的语法形式如下。

```
if (条件表达式):
    语句/语句块
```

其中：

(1) 条件表达式：可以是关系表达式、逻辑表达式、算术表达式等。

(2) 语句/语句块：可以是单个语句，也可以是多个语句。多个语句的缩进必须一致。

当条件表达式的值为真(True)时，执行 if 后的语句(块)，否则不做任何操作，控制将转到 if 语句的结束点。其流程如图 3-2(a)所示。

条件表达式最后被评价为 bool 值 True(真)或 False(假)。如果表达式的结果为数值类型(0)、空字符串("")、空元组(())、空列表([])、空字典({})，其 bool 值为 False(假)，否则其 bool 值为 True(真)。例如，123、"abc"、(1,2)均为 True。

【例 3.2】 单分支结构示例(if_2desc.py)：输入两个数 a 和 b，比较两者的大小，使得 a 大于 b。

```
a = int(input("请输入第 1 个整数:"))
b = int(input("请输入第 2 个整数:"))
print(str.format("输入值:{0}, {1}", a, b))
if (a < b):                                    #a 和 b 交换
    t = a
    a = b
    b = t
print(str.format("降序值:{0}, {1}", a, b))
```

程序运行结果如下。

```
请输入第 1 个整数:23
请输入第 2 个整数:34
输入值:23, 34
降序值:34, 23
```

3.2.3 双分支结构

if 语句双分支结构的语法形式如下。

```
if (条件表达式):
    语句/语句块 1
else:
    语句/语句块 2
```

当条件表达式的值为真(True)时,执行 if 后的语句(块)1,否则执行 else 后的语句(块)2,其流程如图 3-2(b)所示。

Python 提供了下列条件表达式来实现等价于其他语言的三元条件运算符((条件)? 语句 1:语句 2)的功能:

```
条件为真时的值 if (条件表达式) else 条件为假时的值
```

例如,如果 $x \geqslant 0$,则 $y = x$,否则 $y = 0$,可以表述为:

```
y = x  if (x >= 0)  else  0
```

【例 3.3】 计算分段函数:$y=\begin{cases}\sin x+2\sqrt{x+e^4}-(x+1)^3 & x\geqslant 0\\ \ln(-5x)-\dfrac{|x^2-8x|}{7x}+e & x<0\end{cases}$

此分段函数有以下几种实现方式,请读者自行编程测试。

(1) 利用单分支结构实现。

```
if (x>=0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)
if (x<0):
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e
```

(2) 利用双分支结构实现。

```
if (x>=0):
    y = math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)
else:
    y = math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e
```

(3) 利用条件运算语句实现。

```
y = (math.sin(x) + 2 * math.sqrt(x + math.exp(4)) - math.pow(x + 1, 3)) if ((x>=0)) else \
    (math.log(-5 * x) - math.fabs(x * x - 8 * x) / (7 * x) + math.e)
```

3.2.4 多分支结构

if 语句多分支结构的语法形式如下。

```
if (条件表达式 1):
    语句/语句块 1
elif (条件表达式 2):
    语句/语句块 2
...
elif (条件表达式 n):
    语句/语句块 n
[else:
    语句/语句块 n+1;]
```

该语句的作用是根据不同条件表达式的值确定执行哪个语句(块),其流程如图 3-2(c)所示。

【例 3.4】 已知某课程的百分制分数 mark,将其转换为五级制(优、良、中、及格、不及格)的评定等级 grade。评定条件如下:

$$成绩等级=\begin{cases}优 & mark\geqslant 90\\ 良 & 80\leqslant mark<90\\ 中 & 70\leqslant mark<80\\ 及格 & 60\leqslant mark<70\\ 不及格 & mark<60\end{cases}$$

根据评定条件,有以下 3 种方法实现。

方法一:

```
mark = int(input("请输入分数:"))
if (mark >= 90): grade = "优"
elif (mark >= 80): grade = "良"
elif (mark >= 70): grade = "中"
elif (mark >= 60): grade = "及格"
else: grade = "不及格"
```

方法二:

```
if (mark >= 90): grade = "优"
elif (mark >= 80 and mark < 90): grade = "良"
elif (mark >= 70 and mark < 80): grade = "中"
elif (mark >= 60 and mark < 70): grade = "及格"
else: grade = "不及格"
```

方法三:

```
if (mark >= 60): grade = "及格"
elif (mark >= 70): grade = "中"
elif (mark >= 80): grade = "良"
elif (mark >= 90): grade = "优"
else: grade = "不及格"
```

其中,方法一使用关系运算符“>=”,按分数从大到小依次比较;方法二使用关系运算符和逻辑运算符表达完整的条件,即使语句顺序不按比较的分数从大到小依次书写,也可以得到正确的等级评定结果;方法三使用关系运算符“>=”,但按分数从小到大依次比较。

在上述 3 种方法中,方法一、方法二正确,而且方法一简洁明了,方法二虽然正确,但是存在冗余条件。方法三虽然语法没有错误,但是判断结果错误:根据 mark 分数所得等级评定结果只有“及格”和“不及格”两种,请读者根据程序流程自行分析原因。

【例 3.5】 已知坐标点(x,y),判断其所在的象限(if_coordinate.py)。

```
x = int(input("请输入 x 坐标:"))
y = int(input("请输入 y 坐标:"))
if (x == 0 and y == 0): print("位于原点")
elif (x == 0): print("位于 y 轴")
elif (y == 0): print("位于 x 轴")
elif (x > 0 and y > 0): print("位于第一象限")
elif (x < 0 and y > 0): print("位于第二象限")
elif (x < 0 and y < 0): print("位于第三象限")
else: print("位于第四象限")
```

程序运行结果如下。

```
请输入 x 坐标:1
请输入 y 坐标:2
位于第一象限
```

3.2.5　if 语句的嵌套

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套,其一般形式如下。

```
if (条件表达式 1):
    if (条件表达式 11):  ┐
        语句 1           │
    [else:               ├内嵌 if
        语句 2]          ┘
[else:
    if (条件表达式 21):  ┐
        语句 3           │
    [else:               ├内嵌 if
        语句 4]]         ┘
```

【例 3.6】　计算分段函数:$y=\begin{cases}1 & x>0\\0 & x=0\\-1 & x<0\end{cases}$

此分段函数有以下几种实现方式,请读者判断哪些是正确的,并自行编程测试正确的实现方式。

方法一(多分支结构):

```
if (x > 0): y = 1
elif (x == 0): y = 0
else: y = -1
```

方法二(if 语句嵌套结构):

```
if (x >= 0):
    if (x > 0): y = 1
    else: y = 0
else: y = -1
```

方法三:

```
y = 1
if (x != 0):
    if (x < 0): y = -1
else: y = 0
```

方法四:

```
y = 1
if (x != 0):
    if (x < 0): y = -1
    else: y = 0
```

请读者画出每种方法相应的流程图,并进行分析测试。其中,方法一、方法二和方法三是正确的,方法四是错误的。

3.2.6　if 语句的典型示例代码

if 语句的典型示例代码如表 3-1 所示。当 if 或 else 的语句块仅包含一条语句时，该语句也可以直接写在关键字 if 或 else 的同一行后面，以使代码紧凑。

表 3-1　if 语句的典型示例代码

程序功能	代码片段
求绝对值	`if a < 0:` `    a = -a`

a 和 b 按升序排序	`if a > b:`
	`    t = a`
	`    a = b`
	`    b = t`

求 a 和 b 的最大值	`if a > b: maximum = a` `else: maximum = b`
计算两个数相除的余数，如果除数为 0，则给出报错信息	`if b == 0: print("除数为 0")` `else: print("余数为:" + a % b)`

计算并输出一元二次方程的两个根。如果判别式 $b^2-4ac<0$，则显示“方程无实根”的提示信息	`delta = b*b - 4.0*a*c`
	`if delta < 0.0:`
	`    print("方程无实根")`
	`else:`
	`    d = math.sqrt(delta)`
	`    print((-b + d)/(2.0*a))`
	`    print((-b - d)/(2.0*a))`

3.2.7　选择结构综合举例

【例 3.7】　输入 3 个数，按从大到小的顺序排序(if_3desc.py)。

先比较 a 和 b，使得 a>b；然后比较 a 和 c，使得 a>c，此时 a 最大；最后比较 b 和 c，使得 b>c。

```
a = int(input("请输入整数 a:"))
b = int(input("请输入整数 b:"))
c = int(input("请输入整数 c:"))
if (a < b): t = a; a = b; b = t            #使得 a > b
if (a < c): t = a; a = c; c = t            #使得 a > c
if (b < c): t = b; b = c; c = t            #使得 b > c
print("排序结果(降序):", a, b, c)
```

程序运行结果如下。

```
请输入整数 a:3
请输入整数 b:2
请输入整数 c:5
排序结果(降序): 5 3 2
```

【例 3.8】　编程判断某一年是否为闰年(leapyear.py)。判断闰年的条件是年份能被 4 整除但不能被 100 整除，或者能被 400 整除，其判断流程参见图 3-3。

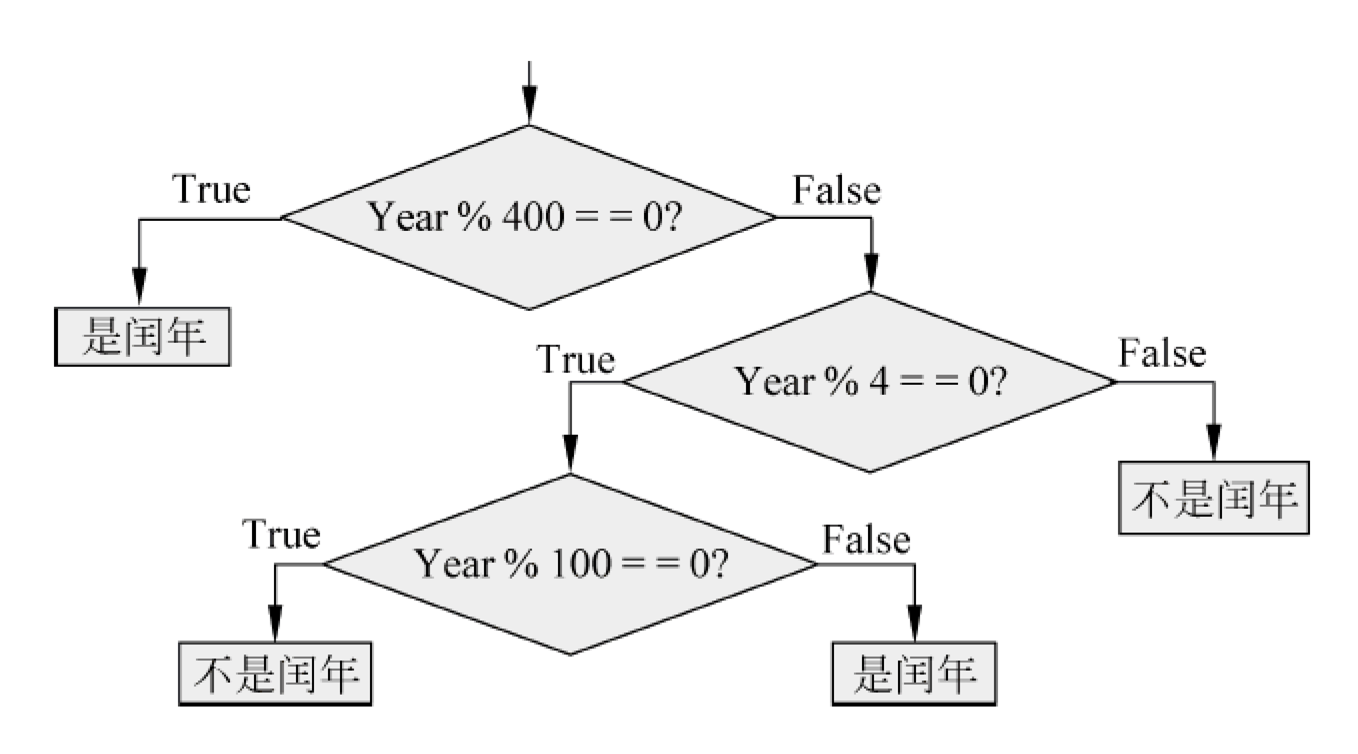


图 3-3　闰年的判断条件

方法一：使用一个逻辑表达式包含所有的闰年条件，相关语句如下。

```
if ((y % 4 == 0 and y % 100 != 0) or y % 400 == 0):
    print("是闰年")
else: print("不是闰年")
```

方法二：使用嵌套的 if 语句，相关语句如下。

```
if (y % 400 == 0): print("是闰年")
else:
    if (y % 4 == 0):
        if (y % 100 == 0): print("不是闰年")
        else: print("是闰年")
    else: print("不是闰年")
```

方法三：使用 if…elif 语句，相关语句如下。

```
if (y % 400 == 0): print("是闰年")
elif (y % 4 != 0): print("不是闰年")
elif (y % 100 == 0): print("不是闰年")
else: print("是闰年")
```

方法四：使用 calendar 模块的 isleap()函数来判断闰年，相关语句如下。

```
if (calendar.isleap(y)): print("是闰年")
else: print("不是闰年")
```

3.3　循环结构

循环结构用来重复执行一条或多条语句，使用循环结构可以减少源程序重复书写的工作量。许多算法需要使用到循环结构，Python 使用 for 语句和 while 语句来实现循环结构。

3.3.1　可迭代对象

可迭代对象(iterable)一次返回一个元素，因此适用于循环。Python()包括以下几种可迭代对象：序列(sequence)，例如字符串(str)、列表(list)、元组(tuple)等；字典(dict)；文件对象；迭代器对象(iterator)；生成器函数(generator)。

迭代器是一个对象，表示可迭代的数据集合，包括方法__iter__()和__next__()，可以实现迭代功能。

生成器是一个函数，使用 yield 语句，每次产生一个值，也可以用于循环迭代。

3.3.2 range 对象

Python 3 中的内置对象 range 是一个迭代器对象，在迭代时产生指定范围的数字序列，其格式如下。

```
range(start, stop[, step])
```

range 返回的数字序列从 start 开始，到 stop 结束(不包含 stop)。如果指定了可选的步长 step，则序列按步长 step 增长。例如：

```
>>> for i in range(1,11): print(i, end = ' ')        #输出:1 2 3 4 5 6 7 8 9 10
>>> for i in range(1,11,3): print(i, end = ' ')      #输出:1 4 7 10
```

注意，Python 2 中 range 的类型为函数，是一个生成器；Python 3 中 range 的类型为类，是一个迭代器。

3.3.3 for 循环

for 语句用于遍历可迭代对象集合中的元素，并对集合中的每个元素执行一次相关的嵌入语句。当集合中的所有元素完成迭代后，控制传递给 for 之后的下一个语句。for 语句的格式如下。

```
for 变量 in 对象集合:
    循环体语句/语句块
```

例如：

```
>>> for i in (1,2,3):
    print(i, i**2, i**3)
1 1 1
2 4 8
3 9 27
```

【例 3.9】 利用 for 循环求 1～100 中所有奇数的和以及所有偶数的和(for_sum1_100.py)。

```
sum_odd = 0; sum_even = 0
for i in range(1, 101):
    if i % 2 != 0:                              #奇数
        sum_odd += i                            #奇数和
    else:                                       #偶数
        sum_even += i                           #偶数和
print("1～100 中所有奇数的和:", sum_odd)
print("1～100 中所有偶数的和:", sum_even)
```

程序运行结果如下。

```
1～100 中所有奇数的和: 2500
1～100 中所有偶数的和: 2550
```

3.3.4 while 循环

与 for 循环一样，while 也是一个预测试的循环，但是 while 在循环开始前并不知道重复执行循环语句序列的次数。while 语句按不同条件执行循环语句(块)零次或多次。while 循环语句的格式如下。

```
while (条件表达式):
    循环体语句/语句块
```

while 循环的执行流程如图 3-4 所示。

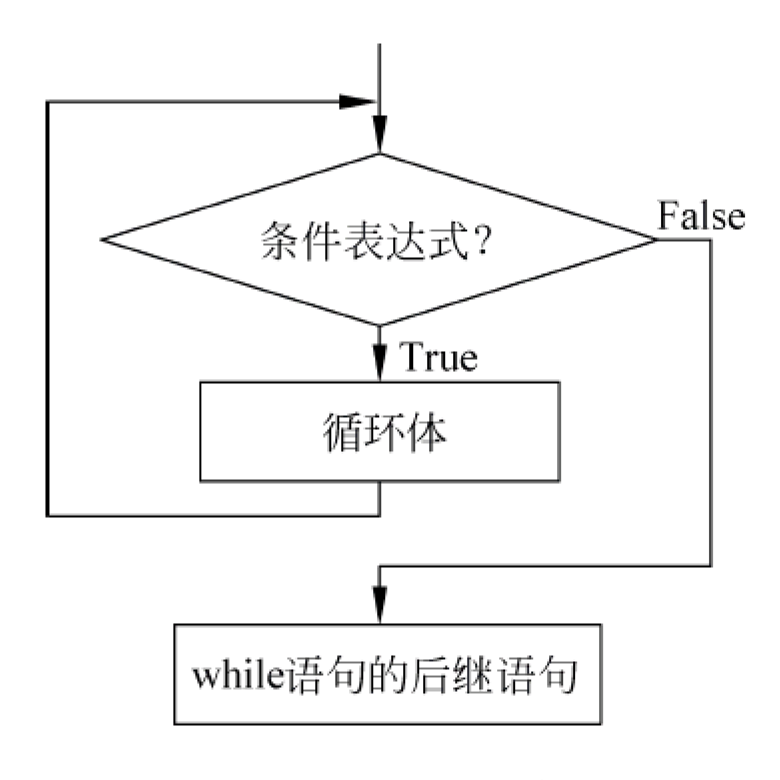


图 3-4　while 循环的执行流程

说明：

(1) while 循环语句的执行过程如下。

① 计算条件表达式。

② 如果条件表达式的结果为 True，控制将转到循环语句（块），即进入循环体。当到达循环语句序列的结束点时转①，即控制转到 while 语句的开始，继续循环。

③ 如果条件表达式的结果为 False，退出 while 循环，即控制转到 while 循环语句的后继语句。

(2) 条件表达式是每次进入循环之前进行判断的条件，可以为关系表达式或逻辑表达式，其运算结果为 True（真）或 False（假）。在条件表达式中必须包含控制循环的变量。


(3) 循环语句序列可以是一条语句，也可以是多条语句。


(4) 在循环语句序列中至少应包含改变循环条件的语句，以使循环趋于结束，避免“死循环”。

【例 3.10】 利用 while 循环求 $\sum_{i=1}^{100} i$，以及 1～100 中所有奇数的和、所有偶数的和（while_sum.py）。

```
i = 1; sum_all = 0; sum_odd = 0; sum_even = 0
while (i <= 100):
    sum_all += i                              #所有数之和
    if (i % 2 == 0):                          #偶数
        sum_even += i                         #偶数和
    else:                                     #奇数
        sum_odd += i                          #奇数和
    i += 1
print("和 = %d、奇数和 = %d、偶数和 = %d" % (sum_all, sum_odd, sum_even))
```

程序运行结果如下。

```
和 = 5050、奇数和 = 2500、偶数和 = 2550
```

【例 3.11】 用以下近似公式求自然对数的底数 e 的值，直到最后一项的绝对值小于 10^{-6} 为止（while_e.py）。

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

```
i = 1; e = 1; t = 1
while (1/t >= pow(10, -6)):
    t *= i
    e += 1 / t
    i += 1
print("e =", e)
```

程序运行结果如下。

```
e = 2.7182818011463845
```

3.3.5　循环的嵌套

若在一个循环体内又包含另一个完整的循环结构，则称之为循环的嵌套。这种语句结构称为多重循环结构。在内层循环中还可以包含新的循环，以形成多层循环结构。

在多层循环结构中两种循环语句（for 循环、while 循环）可以相互嵌套。多重循环的循环次数等于每一重循环次数的乘积。

【例 3.12】 利用嵌套循环打印运行效果如图 3-5 所示的九九乘法表（nest_for.py）。

```
1*1=1  1*2=2  1*3=3  1*4=4  1*5=5  1*6=6  1*7=7  1*8=8  1*9=9
2*1=2  2*2=4  2*3=6  2*4=8  2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1=3  3*2=6  3*3=9  3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1=4  4*2=8  4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1=5  5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1=6  6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1=7  7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1=8  8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*1=9  9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

图 3-5　九九乘法表的运行效果图

```
for i in range(1, 10):                              #外循环
    s = ""
    for j in range(1, 10):                          #内循环
        s += str.format("{0:1}*{1:1}={2:<2} ", i, j, i * j)
    print(s)
```

思考： 请修改程序，分别打印如图 3-6(a)和图 3-6(b)所示的九九乘法表。

```
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12 4*4=16
5*1=5  5*2=10 5*3=15 5*4=20 5*5=25
6*1=6  6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7  7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8  8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9  9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

(a) 下三角

```
1*1=1  1*2=2  1*3=3  1*4=4  1*5=5  1*6=6  1*7=7  1*8=8  1*9=9
       2*2=4  2*3=6  2*4=8  2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
              3*3=9  3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
                     4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
                            5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
                                   6*6=36 6*7=42 6*8=48 6*9=54
                                          7*7=49 7*8=56 7*9=63
                                                 8*8=64 8*9=72
                                                        9*9=81
```

(b) 上三角

图 3-6　九九乘法表的另外两种显示效果

3.3.6　break 语句

break 语句用于退出 for、while 循环，即提前结束循环，接着执行循环语句的后继语句。注意，当多个 for、while 语句彼此嵌套时，break 语句只应用于最里层的语句，即 break 语句只能跳出最近的一层循环。

【例 3.13】 使用 break 语句中止循环（break.py）。

```
while True:
    s = input('请输入字符串(按 Q 或者 q 键结束):')
    if s.upper() == 'Q':
        break
    print('字符串的长度为:', len(s))
```

程序运行结果如下。

```
请输入字符串(按 Q 或者 q 键结束):Hello, World!
字符串的长度为: 13
请输入字符串(按 Q 或者 q 键结束):您好!
字符串的长度为: 3
请输入字符串(按 Q 或者 q 键结束):q
```

【例 3.14】 编程判断所输入的任意一个正整数是否为素数(prime1.py 和 prime2.py)。

所谓素数(或称质数)，是指除了 1 和该数本身之外不能被任何整数整除的正整数。判断一个正整数 m 是否为素数，只要判断 m 可否被 2～$\sqrt{m}$中的任何一个整数整除即可，如果 m 不能被此范围中的任何一个整数整除，m 即为素数，否则 m 为合数。

方法一(利用 for 循环和 break 语句)：

```
import math
m = int(input("请输入一个整数(>1):"))
k = int(math.sqrt(m))
for i in range(2, k + 2):
    if m % i == 0:
        break                               #可以整除，肯定不是素数，结束循环
if i == k+1 : print(m, "是素数!")
else: print(m, "是合数!")
```

方法二(利用 while 循环和 bool 变量)：

```
import math
m = int(input("请输入一个整数(>1):"))
k = int(math.sqrt(m))
flag = True                                 #先假设所输整数为素数
i = 2
while (i <= k and flag == True):
    if (m % i == 0): flag = False           #可以整除，肯定不是素数，结束循环
    else: i += 1
if (flag == True): print(m, "是素数!")
else: print(m, "是合数!")
```

3.3.7 continue 语句

continue 语句类似于 break 语句，也必须在 for、while 循环中使用，但它结束本次循环，即跳过循环体内 continue 下面尚未执行的语句，返回到循环的起始处，并根据循环条件判断是否执行下一次循环。

continue 语句和 break 语句的区别在于：continue 语句仅结束本次循环，并返回到循环的起始处，如果循环条件满足就开始执行下一次循环；而 break 语句则是结束循环，跳转到循环的后继语句执行。

与 break 语句类似，当多个 for、while 语句彼此嵌套时 continue 语句只应用于最里层的语句。

【例 3.15】 使用 continue 语句跳过循环示例(continue_score.py)。要求输入若干学生成绩(按 Q 或 q 键结束)，如果成绩<0，则重新输入。统计学生人数和平均成绩。

```
num = 0; scores = 0;                        #初始化学生人数和成绩之和
while True:
    s = input('请输入学生成绩(按 Q 或 q 键结束):')
    if s.upper() == 'Q':
        break
    if float(s) < 0:                        #成绩必须≥0
        continue
    num += 1                                #统计学生人数
    scores += float(s)                      #计算成绩之和
print('学生人数为:{0},平均成绩为:{1}'.format(num,scores / num))
```

程序运行结果如下。

```
请输入学生成绩(按 Q 或 q 键结束):65
请输入学生成绩(按 Q 或 q 键结束):87
请输入学生成绩(按 Q 或 q 键结束):-40
请输入学生成绩(按 Q 或 q 键结束):q
学生人数为:2,平均成绩为:76.0
```

【例 3.16】　显示 100～200 不能被 3 整除的数(continue_div3.py)。要求一行显示 10 个数。程序运行结果如图 3-7 所示。

```
100~200之间不能被3整除的数为:
100  101  103  104  106  107  109  110  112  113
115  116  118  119  121  122  124  125  127  128
130  131  133  134  136  137  139  140  142  143
145  146  148  149  151  152  154  155  157  158
160  161  163  164  166  167  169  170  172  173
175  176  178  179  181  182  184  185  187  188
190  191  193  194  196  197  199  200
```

图 3-7　显示 100～200 不能被 3 整除的数

```
j = 0                                        #控制一行显示的数值个数
print('100～200 之间不能被 3 整除的数为:')
for i in range(100, 200 + 1):
    if (i % 3 == 0): continue                #跳过能被 3 整除的数
    print(str.format("{0:<5}",i), end = "")  #每个数占 5 个位置,不足的后面加空格,并且不
                                             #换行
    j += 1
    if (j % 10 == 0): print()                #一行显示 10 个数后换行
```

3.3.8　死循环

如果 while 循环结构中的循环控制条件一直为真,则循环将无限继续,程序将一直运行下去,从而形成死循环。

当程序死循环时会造成程序没有任何响应,或者造成不断输出(例如控制台输出、文件写入、打印输出等)。

在程序的循环体中,插入调试输出语句,可以判断程序是否为死循环。注意,有的程序算法十分复杂,可能需要运行很长时间,但并不是死循环。

在大多数计算机系统中,可以使用 Ctrl+C 组合键中止当前程序的运行。

【例 3.17】　死循环示例(infinite.py)。

```
import math
while True:                                  #循环条件一直为真
   num = float(input("请输入一个正数:"))
   print(str(num), "的平方根为:", math.sqrt(num))
print("Good bye!")
```

本程序因为循环条件为“while True”,所以将一直重复提示用户输入一个正数,计算并输出该数的平方根,从而形成死循环。所以,最后的“print("Good bye!")”语句将没有机会执行。

3.3.9　else 子句

for、while 语句可以附带一个 else 子句(可选)。如果 for、while 语句没有被 break 语句中止,则会执行 else 子句,否则不执行。其语法如下。

```
for 变量 in 对象集合:
   循环体语句(块)1
else:
  语句(块)2
```

或者:

```
while (条件表达式):
   循环体语句(块)1
else:
  语句(块)2
```

【例 3.18】 使用 for 语句的 else 子句(for_else.py)。

```
hobbies = ""
for i in range(1, 3 + 1):
    s = input('请输入爱好之一(最多 3 个,按 Q 或 q 键结束):')
    if s.upper() == 'Q':
         break
    hobbies += s + ''
else:
    print('您输入了 3 个爱好.')
print('您的爱好为:', hobbies)
```

程序运行结果如下。

```
>>>
请输入爱好之一(最多 3 个,按 Q 或 q 键结束):旅游
请输入爱好之一(最多 3 个,按 Q 或 q 键结束):音乐
请输入爱好之一(最多 3 个,按 Q 或 q 键结束):运动
您输入了 3 个爱好.
您的爱好为: 旅游 音乐 运动
>>>
请输入爱好之一(最多 3 个,按 Q 或 q 键结束):音乐
请输入爱好之一(最多 3 个,按 Q 或 q 键结束):q
您的爱好为: 音乐
```

3.3.10 enumerate()函数和循环

Python 语言的 for 循环直接迭代对象集合中的元素,如果需要在循环中使用索引下标访问集合元素,则可以使用内置的 enumerate()函数。

enumerate()函数用于将一个可遍历的数据对象(例如列表、元组或字符串)组合为一个索引序列,并返回一个可迭代对象,故在 for 循环当中可直接迭代下标和元素。

【例 3.19】 enumerate()函数和下标元素循环示例(enumerate.py)。

```
seasons = ['Spring', 'Summer', Autumn, 'Winter']
for i, s in enumerate(seasons, start = 1):        #start 默认从 0 开始
    print("第{0}季节:{1}".format(i, s))
```

程序运行结果如下。

```
第 1 季节:Spring
第 2 季节:Summer
第 3 季节:Autumn
第 4 季节:Winter
```

3.3.11 zip()函数和循环

如果需要并行遍历多个可迭代对象,则可以使用 Python 的内置函数 zip()。

zip()函数将多个可迭代对象中对应的元素打包成一个个元组,然后返回一个可迭代对

象。如果元素的个数不一致,则返回列表的长度与最短的对象相同。

利用运算符 * 还可以实现将元组解压为列表。例如:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zip(x, y)                              #输出:<zip object at 0x000001E663DEB988>
>>> list(zip(x, y))                        #输出:[(1, 4), (2, 5), (3, 6)]
>>> a, b = zip( * zip(x, y))
>>> a, b                                   #输出:((1, 2, 3), (4, 5, 6))
```

【例 3.20】 zip()函数和并行循环示例(zip.py)。

```
evens = [0, 2, 4, 6, 8]
odds = [1, 3, 5, 7, 9]
for e, o in zip(evens, odds):
    print("{0} * {1} = {2}".format(e, o, e * o))
```

程序运行结果如下。

```
0 * 1 = 0
2 * 3 = 6
4 * 5 = 20
6 * 7 = 42
8 * 9 = 72
```

3.3.12　map()函数和循环

如果需要遍历可迭代对象,并使用指定函数处理对应的元素,则可以使用 Python 的内置函数 map()。

map(func, seq1[, seq2,…])函数将 func 作用于 seq 中的每一个元素,并将所有的调用结果作为可迭代对象返回。如果 func 为 None,该函数的作用等同于 zip()函数。

例如,如果要返回列表中每个字符串的长度,可以使用内置的 map()函数和 len()函数:

```
>>> map(len, ['Spring', 'Summer', 'Fall', 'Winter'])
<map object at 0x000001C512590F98>
>>> list(map(len, ['Spring', 'Summer', 'Fall', 'Winter']))   #输出:[6, 6, 4, 6]
```

【例 3.21】 map()函数和循环示例。

```
>>> list(map(abs, [-1, 0, 7, -8]))          #计算绝对值.输出:[1, 0, 7, 8]
>>> list(map(pow, range(5), range(5)))      #计算乘幂.输出:[1, 1, 4, 27, 256]
>>> list(map(ord,'abcdef'))                 #计算 ASCII 码.输出:[97, 98, 99, 100, 101, 102]
>>> list(map(lambda x,y:x+y,'abc','de'))    #字符串拼接.输出:['ad', 'be']
```

3.3.13　循环语句的典型示例代码

使用 for 语句和 while 语句都能实现循环功能,选择哪种语法构造取决于程序员的偏好。循环语句的典型示例如表 3-2 所示。

表 3-2　for 语句和 while 语句的典型示例

功能示例	实现代码
输出 n 个数(0～n－1)的 2 的乘幂的值列表	power = 1 for i in range(n): print(str(i) + " " + str(power)) power * = 2

续表

功 能 示 例	实 现 代 码
输出小于或等于 n 的最大的 2 的乘幂的值	power = 1 while 2 * power <= n: power *= 2 print(power)
计算并输出 1+2+…+n 的和	total = 0 for i in range(1, n+1): total += i print(total)
计算并输出 n 的阶乘（n!＝1×2×…×n）	factorial = 1 for i in range(1, n+1): factorial *= i print(factorial)
输出半径为 1～n 的圆的周长列表	for r in range(1, n+1): print("r=" + str(r), end=" ") print("p=" + str(2.0 * math.pi * r))

3.3.14 循环结构综合举例

【例 3.22】 使用牛顿迭代法求解平方根（sqrt.py）。其运行效果如图 3-8 所示。

计算一个正实数 a 的平方根可以使用牛顿迭代法实现：首先假设 t＝a，开始循环，如果 t＝a/t（或小于容差），则 t 等于 a 的平方根，循环结束并返回结果；否则将 t 和 a/t 的平均值赋给 t，继续循环。

```
EPSILON = 1e-15                              #容差
a = float(input("请输入正实数 a:"))           #正实数 a
t = a                                        #假设平方根 t=a
while abs(t - a/t) > (EPSILON * t):
    t = (a/t + t) / 2.0                      #将 t 和 a/t 的平均值赋给 t
print(t)                                     #输出 a 的平方根
```

【例 3.23】 显示 Fibonacci 数列（for_fibonacci.py）：1、1、2、3、5、8……的前 20 项。即

$$\begin{cases} F_1=1 & n=1 \\ F_2=1 & n=2 \\ F_n=F_{n-1}+F_{n-2} & n\geqslant 3 \end{cases}$$

要求每行显示 4 项。其运行效果如图 3-9 所示。

```
请输入正实数a: 2
1.414213562373095
```

图 3-8 使用牛顿迭代法求解平方根

```
   1     1     2     3
   5     8    13    21
  34    55    89   144
 233   377   610   987
1597  2584  4181  6765
```

图 3-9 显示 Fibonacci 数列

相关语句如下：

```
f1 = 1; f2 = 1
for i in range(1, 11):
```

```
print(str.format("{0:6}{1:6}", f1, f2), end = " ") #每次输出两个数,每个数占 6 位,以空格分隔
if i % 2 == 0: print()                             #显示 4 项后换行
f1 += f2; f2 += f1
```

3.4 复 习 题

一、选择题

1. 下面的 Python 循环体的执行次数与其他不同的是________。

A.
```
i = 0
while(i <= 10):
    print(i)
    i = i + 1
```
B.
```
i = 10
while(i > 0):
    print(i)
    i = i - 1
```
C.
```
for i in range(10):
    print(i)
```
D.
```
for i in range(10, 0, -1):
    print(i)
```

2. 执行下列 Python 语句将产生的结果是________。

```
x = 2; y = 2.0
if(x == y): print("Equal")
else: print("Not Equal")
```

A. Equal　　B. Not Equal　　C. 编译错误　　D. 运行时错误

3. 执行下列 Python 语句将产生的结果是________。

```
i = 1
if (i): print(True)
else: print(False)
```

A. 输出 1　　B. 输出 True　　C. 输出 False　　D. 编译错误

4. 用 if 语句表示如下分段函数 f(x),下面程序不正确的是________。

$$f(x)=\begin{cases}2x+1 & x\geqslant 1\\ 3x/(x-1) & x<1\end{cases}$$

A.
```
if(x >= 1):f = 2 * x + 1
f = 3 * x/(x - 1)
```
B.
```
if (x >= 1): f = 2 * x + 1
if (x < 1): f = 3 * x/(x - 1)
```
C.
```
f = 2 * x + 1
if (x < 1): f = 3 * x/(x - 1)
```
D.
```
if (x < 1): f = 3 * x/(x - 1)
else: f = 2 * x + 1
```

5. 下面的 if 语句统计满足"性别(gender)为男、职称(rank)为教授、年龄(age)小于 40 岁"条件的人数,正确的语句为________。

A. if (gender == "男" or age < 40 and rank == "教授"): n+=1

B. if (gender =="男" and age < 40 and rank=="教授"): n+=1

C. if (gender =="男" and age < 40 or rank=="教授"): n+=1

D. if (gender =="男" or age < 40 or rank=="教授"): n+=1

6. 下面的程序段求 x 和 y 两个数中的大数,________是不正确的。

A. `maxNum = x if x > y else y`

B. `maxNum = math.max(x, y)`

C.
```
if (x > y): maxNum = x
else: maxNum = y
```
D.
```
if (y >= x): maxNum = y
maxNum = x
```

7. 下面的 if 语句统计"成绩(score)优秀的男生以及不及格的男生"的人数,正确的语句

为________。

A. if (gender=="男" and score<60 or score>=90): n+=1

B. if (gender =="男" and score<60 and score>=90): n+=1

C. if (gender =="男" and (score<60 or score>=90)): n+=1

D. if (gender =="男" or score<60 or score>=90): n+=1

8. 用 if 语句表示如下分段函数：

$$y=\begin{cases} x^2-2x+3 & x<1 \\ \sqrt{x-1} & x\geqslant 1 \end{cases}$$

下面程序段不正确的是________。

A.
```
if (x < 1): y = x * x - 2 * x+3
else: y = math.sqrt(x - 1)
```

B.
```
if (x < 1): y = x * x - 2 * x + 3
y = math.sqrt(x - 1)
```

C.
```
y = x * x - 2 * x+3
if (x >= 1): y = math.sqrt(x - 1)
```

D.
```
if (x < 1): y = x * x - 2 * x + 3
if (x >= 1): y = math.sqrt(x - 1)
```

9. 在以下 for 语句结构中，________不能完成 1～10 的累加功能。

A. for i in range(10,0): total += i

B. for i in range(1,11): total += i

C. for i in range(10,0,-1): total += i

D. for i in (10,9,8,7,6,5,4,3,2,1): total += i

二、填空题

1. 迭代器是一个对象，表示可迭代的数据集合，包括方法________和________，可实现迭代功能。

2. 在 Python 无穷循环 while True: 的循环体中可以使用________语句退出循环。

3. Python 语句“for i in range(1,21,5): print(i, end=' ')”的输出结果为________。

4. Python 语句“for i in range(10,1,-2): print(i, end=' ')”的输出结果为________。

5. 循环语句 for i in range(-3,21,4)的循环次数为________。

6. 要使语句 for i in range(__,-4,-2)循环执行 15 次，则循环变量 i 的初值应当为________。

7. 执行下列 Python 语句后的输出结果是________，循环执行了________次。

```
i = -1;
while (i < 0): i * = i
print(i)
```

三、思考题

1. 说明以下 3 个 if 语句的区别：

(1)
```
if (i > 0):
        if (j > 0): n = 1
        else: n =2
```

(2)
```
if (i > 0):
        if (j > 0):n = 1
else: n = 2
```

(3)
```
if (i > 0): n = 1
else:
    if (j > 0): n = 2
```

2. 下列 Python 语句的运行结果为________。

```
for i in range(3): print(i, end = ' ')
for i in range(2,5): print(i, end = ' ')
```

3. 阅读下面的 Python 程序，请问程序的功能是什么？

```
import math; n = 0
for m in range(101, 201, 2):
    k = int(math.sqrt(m))
    for i in range(2, k + 2):
        if m % i == 0: break
    if i == k + 1:
        if n % 10 == 0:print()
        print('%d' % m, end = ' ')
        n += 1
```

4. 阅读下面的 Python 程序，请问输出结果是什么？

```
n = int(input("请输入图形的行数:"))
for i in range(0, n):
      for j in range(0, 10 - i): print(" ",end = ' ')
      for j in range(0, 2 * i + 1): print(" * ", end = ' ')
      print("\n")
```

5. 阅读下面的 Python 程序，请问输出结果是什么？程序的功能是什么？

```
from math import *
print("三位数中所有的水仙花数为:")
for i in range(100,1000):
      n1 = i//100; n2 = (i % 100)//10; n3 = i % 10
      if(pow(n1,3) + pow(n2,3) + pow(n3,3) == i): print(i, end = ' ')
```

6. 阅读下面的 Python 程序，请问输出结果是什么？程序的功能是什么？

```
print("1～1000 所有的完数有,其因子为:")
for n in range(1,1001):
      total = 0;j = 0;factors = []
      for i in range(1,n):
            if(n % i == 0):
                  factors.append(i); total += i
      if(total == n): print("{0}: {1}".format(n, factors))
```

7. 阅读下面的 Python 程序，请问输出结果是什么？程序的功能是什么？

```
m = int(input("请输入整数 m:")); n = int(input("请输入整数 n:"))
while(m!= n):
    if (m > n): m = m - n
    else: n = n - m
print(m)
```

3.5 上机实践

1. 完成本章中的例 3.1～例 3.23，熟悉 Python 语言的 3 种基本控制结构，即顺序结构、选择结构、循环结构。

2. 编写程序，计算 1＋2＋3＋…＋100 之和。

3. 编写程序，计算 10＋9＋8＋…＋1 之和。

4. 编写程序,计算 1+3+5+7…+99 之和。

5. 编写程序,计算 2+4+6+8…+100 之和。

6. 编写程序,使用不同的实现方法输出 2000~3000 的所有闰年,运行效果如图 3-10 所示。

```
2000 2004 2008 2012 2016 2020 2024 2028 2032 2036 2040 2044 2048 2052 2056 2060 2064 2068
2072 2076 2080 2084 2088 2092 2096 2104 2108 2112 2116 2120 2124 2128 2132 2136 2140 2144
2148 2152 2156 2160 2164 2168 2172 2176 2180 2184 2188 2192 2196 2204 2208 2212 2216 2220
2224 2228 2232 2236 2240 2244 2248 2252 2256 2260 2264 2268 2272 2276 2280 2284 2288 2292
2296 2304 2308 2312 2316 2320 2324 2328 2332 2336 2340 2344 2348 2352 2356 2360 2364 2368
2372 2376 2380 2384 2388 2392 2396 2400 2404 2408 2412 2416 2420 2424 2428 2432 2436 2440
2444 2448 2452 2456 2460 2464 2468 2472 2476 2480 2484 2488 2492 2496 2504 2508 2512 2516
2520 2524 2528 2532 2536 2540 2544 2548 2552 2556 2560 2564 2568 2572 2576 2580 2584 2588
2592 2596 2604 2608 2612 2616 2620 2624 2628 2632 2636 2640 2644 2648 2652 2656 2660 2664
2668 2672 2676 2680 2684 2688 2692 2696 2704 2708 2712 2716 2720 2724 2728 2732 2736 2740
2744 2748 2752 2756 2760 2764 2768 2772 2776 2780 2784 2788 2792 2796 2800 2804 2808 2812
2816 2820 2824 2828 2832 2836 2840 2844 2848 2852 2856 2860 2864 2868 2872 2876 2880 2884
2888 2892 2896 2904 2908 2912 2916 2920 2924 2928 2932 2936 2940 2944 2948 2952 2956 2960
2964 2968 2972 2976 2980 2984 2988 2992 2996
```

图 3-10 2000~3000 的所有闰年

7. 编写程序,计算 $S_n=1-3+5-7+9-11+\cdots$。

提示:

可以使用 if i%2==0 的语句形式判断 i 是否为偶数。

8. 编写程序,计算 $S_n=1+1/2+1/3+\cdots$。

9. 编写程序,打印九九乘法表。要求输出九九乘法表的各种显示效果(上三角、下三角、矩形块等方式)。

10. 编写程序,输入三角形的 3 条边,先判断是否可以构成三角形,如果可以,则进一步求三角形的周长和面积,否则报错"无法构成三角形!"。其运行效果如图 3-11 所示(结果均保留一位小数)。

提示:

(1) 3 个数可以构成三角形必须满足如下条件:每条边的边长均大于 0,并且任意两边之和大于第三边。

(2) 已知三角形的 3 条边,则三角形的面积 $=\sqrt{h*(h-a)*(h-b)*(h-c)}$,其中 h 为三角形周长的一半。

11. 编写程序,输入 x,根据如下公式计算分段函数 y 的值。请分别利用单分支语句、双分支结构以及条件运算语句等方法实现。其运行效果如图 3-12 所示。

$$y=\begin{cases}\dfrac{x^2-3x}{x+1}+2\pi+\sin x & x\geqslant 0\\ \ln(-5x)+6\sqrt{|x|+e^4}-(x+1)^3 & x<0\end{cases}$$

```
请输入三角形的边A:    3
请输入三角形的边B:    4
请输入三角形的边C:    5
三角形三边分别为:  a=3.0, b=4.0, c=5.0
三角形的周长 = 12.0, 面积 = 6.0
```

图 3-11 三角形周长和面积的运行效果

```
请输入x:    -1
方法一: x = -1.0, y = 46.34793812414429
方法二: x = -1.0, y = 46.34793812414429
方法三: x = -1.0, y = 46.34793812414429
```

图 3-12 分段函数的运行效果

12. 编写程序,输入一元二次方程的 3 个系数 a、b 和 c,求 $ax^2+bx+c=0$ 方程的解。其运行效果如图 3-13 所示。

提示:

(1) 方程 $ax^2+bx+c=0$ 的解有以下几种情况。

① a=0 and b=0,无解。

```
请输入系数a:    0
请输入系数b:    0
请输入系数c:    6
此方程无解!
```
(a) 无解

```
请输入系数a:    0
请输入系数b:    1
请输入系数c:    2
此方程的解为:  -2.0
```
(b) 一个实根

```
请输入系数a:    1
请输入系数b:    -2
请输入系数c:    1
此方程有两个相等实根:  1.0
```
(c) 两个相等实根

```
请输入系数a:    1
请输入系数b:    -1
请输入系数c:    -6
此方程有两个不等实根:  3.0 和 -2.0
```
(d) 两个不等实根

```
请输入系数a:    1
请输入系数b:    -1
请输入系数c:    0.5
此方程有两个不等实根:  0.5+0.5i 和 0.5-0.5i
```
(e) 两个共轭复根

图 3-13　求解一元二次方程

② a=0 and b!=0,有一个实根：$x=-\frac{c}{b}$。

③ $b^2-4ac=0$,有两个相等实根：$x_1=x_2=-\frac{b}{2a}$。

④ $b^2-4ac>0$,有两个不等实根：$x_1=-\frac{b}{2a}+\frac{\sqrt{b^2-4ac}}{2a}$,$x_2=-\frac{b}{2a}-\frac{\sqrt{b^2-4ac}}{2a}$。

⑤ $b^2-4ac<0$,有两个共轭复根：$x_1=-\frac{b}{2a}+\frac{\sqrt{4ac-b^2}}{2a}i$,$x_2=-\frac{b}{2a}-\frac{\sqrt{4ac-b^2}}{2a}i$。

(2) 可以利用“print(str.format("此方程有两个不等虚根：{0}+{1}i 和 {0}-{1}i　", realPart, imagPart))”的语句形式输出方程的两个共轭复根。

13. 编写程序,输入整数 n(n≥0),分别利用 for 循环和 while 循环求 n!。其运行效果如图 3-14 所示。

提示：

(1) n! = n×(n-1)×(n-2)×…×2×1。例如 5!=5×4×3×2×1=120,特别地,0!=1。

(2) 一般情况下,累乘的初值为 1,累加的初值为 0。

(3) 如果输入的是负整数,则继续提示输入非负整数,直到 n≥0。

14. 编写程序,产生两个 0~100(包含 0 和 100)的随机整数 a 和 b,求这两个整数的最大公约数和最小公倍数。其运行效果如图 3-15 所示。

```
请输入非负整数n: -5
请输入非负整数n: 5
   for循环: 5! = 120
while循环: 5! = 120
```

图 3-14　阶乘的运行效果

```
整数1 = 88, 整数2 = 16
最大公约数 = 8, 最小公倍数 = 176
```

图 3-15　最大公约数和最小公倍数的运行效果

提示：

(1) 可以利用“random.randint(0,100)”的语句形式生成 0~100(包含 0 和 100)的随机整数。

(2) 利用“辗转相除法”求最大公约数,具体算法如下。

① 对于已知的两个正整数 m、n,使得 m>n。

② m 除以 n 得余数 r。

③ 若 r≠0,则令 m←n,n←r,继续相除得到新的余数 r。若仍然 r≠0,则重复此过程,直到 r=0 为止。最后的 m 就是最大公约数。

(3) 求得了最大公约数,最小公倍数就是已知的两个正整数之积除以最大公约数的商。

3.6 案例研究：使用嵌套循环实现图像处理算法

在科学计算和各种算法中经常需要使用嵌套循环来处理数据。

例如，图像在计算机中是由像素点组成的二维数组，每个像素点的位置被表示为两个整数的元组，像素的值根据图像模式由对应的元组组成（例如 RGB 模式表示为 3 个整数值组成的元组，分别表示构成颜色的红、蓝、绿的值，范围为 0 到 255）。

图像处理（例如复制、旋转、裁剪和平滑图像等）的算法根本上就是使用嵌套循环模式对这些像素进行处理。

本章案例研究使用 Python 第三方图像处理库 Pillow 中 PIL.Image 模块的 Image 类的方法 getpixel()和 putpixel()来读取和修改特定位置（loc）处的像素的颜色值（pix），然后使用嵌套循环实现图像处理的基本算法，目的是使学生深入了解 Python 数据结构和基本算法流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 4 章

常用内置数据类型


视频讲解

计算机能处理各种类型的数据，包括数值、文本等，不同的数据属于不同的数据类型，支持不同的运算和操作。

Python 语言提供了丰富的内置数据类型，用于有效地处理各种类型的数据。本章将主要讨论 4 种 Python 内置数据类型，即 int（整型）、float（浮点型）、bool（布尔型）和 str（字符串）。后续章节将讨论其他内置数据类型和自定义数据类型。

4.1 Python 内置数据类型概述

在 Python 语言中一切皆为对象，而每个对象属于某个数据类型。Python 的数据类型包括内置的数据类型、模块中定义的数据类型和用户自定义的类型。

通过字面量或调用对象的构造方法可以创建数据类型的实例对象，然后使用运算符、内置函数、系统函数和对象方法进行运算操作。

4.1.1 数值数据类型

Python 包括 4 种内置的数值类型。

(1) 整数类型(int)：用于表示整数。例如，123、1024、−982。

(2) 布尔类型(bool)：用于表示布尔逻辑值。例如，True、False。

(3) 浮点类型(float)：用于表示实数。例如，3.14、−1.23、1.1E10、−3e−4。

(4) 复数类型(complex)：用于表示复数。例如，3+4j、−2−4j、1.2+3.4j。

数值可以使用运算符（四则运算 +、−、*、/ 以及幂运算 ** 等）、内置函数（abs()、round() 等）、math/cmath 模块中的数学函数、int/float/complex/bool 类的方法。

4.1.2 序列数据类型

序列数据类型表示若干有序数据。Python 序列数据类型分为不可变序列数据类型和可变序列数据类型。

不可变序列数据类型包括以下 3 种。

(1) 字符串(str)：表示 Unicode 字符序列。例如，"hello"。

(2) 元组类型(tuple)：表示任意类型数据的序列。例如，(1, 2, 3)，(1, "2")。

(3) 字节序列(bytes)：表示字节（8 位）序列数据。例如，b'abc'。

可变序列数据类型包括以下两种。

(1) 列表类型(list)：表示可以修改的任意类型数据的序列。例如，[1, "two"]。

(2) 字节数组(bytearray):表示可以修改的字节(8位)数组。

4.1.3 集合数据类型

集合数据类型表示若干数据的集合,数据项目没有顺序,且不重复。Python集合数据类型包括以下两种。

(1) 集(set):可变对象。例如,{1, 2, 3}。

(2) 不可变集(frozenset):不可变对象。例如:

```
>>> frozenset('abc')             #输出:frozenset({'a', 'c', 'b'})
```

4.1.4 字典数据类型

字典数据类型用于表示键/值对的字典。Python内置的字典数据类型为dict。例如,{1: "one", 2: "two"}。

4.1.5 NoneType、NotImplementedType和EllipsisType

Python包含3种特殊的数据类型,即NoneType、NotImplementedType和EllipsisType。

1. NoneType

NoneType数据类型包含唯一值None,主要用于表示空值,如没有返回值的函数的结果。例如:

```
>>> None
>>> type(None)                   #输出:<class 'NoneType'>
```

2. NotImplementedType

NotImplementedType数据类型包含唯一值NotImplemented。在进行数值运算和比较运算时,如果对象不支持,则可能返回该值。例如:

```
>>> NotImplemented               #输出:NotImplemented
>>> type(NotImplemented)         #输出:<class 'NotImplementedType'>
```

3. EllipsisType

EllipsisType数据类型包含唯一值Ellipsis,表示省略字符串符号"…"。例如:

```
>>> Ellipsis                     #输出:Ellipsis
>>> type(Ellipsis)               #输出:<class 'ellipsis'>
```

4.1.6 其他数据类型

Python中的一切对象都有一个数据类型,模块、类、对象、函数都属于某种数据类型。

Python解释器包含内置类型,例如代码对象(Code objects)、框架对象(Frame objects)、跟踪对象(Traceback objects)、切片对象(Slice objects)、静态方法对象(Static method objects)、类方法对象(Class method objects)。这部分涉及Python语言本身的构造。

4.2 int类型

整数数据类型(int)是表示整数的数据类型。与其他计算机语言有精度限制不同,Python中的整数位数可以为任意长度(只受限制于计算机内存)。整型对象是不可变对象。

4.2.1　整型字面量

数字字符串(前面可以带负号“－”)即整型字面量。Python 解释器自动创建 int 型对象实例。

数字字符串通常被解释为十进制(基数为 10),可以用前缀表示其他进制的整数,但跟在前缀后面的数字必须适合于数制。整型字面量如表 4-1 所示。

表 4-1　整型字面量

数　制	前　缀	基 本 数 码	示　例
十进制(以 10 为基)		0～9	0、1、2、7、999、－12(负数)、＋12(正数)
十六进制(以 16 为基)	0x(或 0X)	0～9 和 A～F(或 a～f)	0x0、0X1、0x2、0X7、0x3e7
八进制(以 8 为基)	0o(或 0O)	0～7	0o0、0O1、0o2、0O7、0o1747
二进制(以 2 为基)	0b(或 0B)	0～1	0b0、0B1、0b10、0B111、0b1100011

【例 4.1】　整型字面量示例。

```
>>> a = 123
>>> type(a)                             #输出:<class 'int'>
>>> 1_000_000_000_000_000               #输出:1000000000000000
>>> 0x_FF_FF_FF_FF                      #输出:4294967295
```

说明: Python 3.7 支持使用下画线作为整数或者浮点数的千分位标记,以增强大数值的可阅读性。二进制、八进制、十六进制则使用下画线区分 4 位标记。

4.2.2　int 对象

int 是 Python 内置的数据类型,用户可以创建 int 类型的对象实例,其基本形式如下。

```
int(x = 0)                              #创建 int 对象(十进制)
int(x, base = 10)                       #创建 int 对象,指定进制为 base(2～36)
```

通过创建 int 对象可以把数值或任何符合格式的字符串或其他对象转换为 int 对象。

注意: 如果对象 x 不能转换为整型,则导致 TypeError; 如果对象 x 转换失败,则导致 ValueError。

【例 4.2】　int 对象示例。

```
>>> int                                 #输出:<class 'int'>
>>> int(), int(123), int('456'), int(1.23)#输出:(0, 123, 456, 1)
>>> int('FF', 16), int('100', 2)        #输出:(255, 4)
>>> int('abc')                          #报错.ValueError: invalid literal for int() with base
                                        #10: 'abc'
>>> int(100, 2)                         #报错.TypeError: int() can't convert non-string with
                                        #explicit base
```

4.2.3　int 对象的方法

int 对象 i 包含的主要方法如下。

```
i.bit_length()  #返回 i 的二进制位数,不包括符号
```

【例 4.3】　int 对象方法示例。

```
>>> i = -10
```

```
>>> bin(i)                                    #数值转换为二进制字符串.输出:'-0b1010'
>>> i.bit_length(), int.bit_length(i)         #返回 i 的二进制位数.输出:(4, 4)
```

4.2.4　整数的运算

整数对象支持关系运算、算术运算、位运算符、内置函数、math 模块中的数学运算函数以及 int 对象方法(参见 4.2.3 节)等运算操作。

在 Python 语言中,常用的 int 数据类型对象的运算表达式如表 4-2 所示。

表 4-2　常用的 int 数据类型表达式

表　达　式	结　　果	说　　明
123	123	整数字面值
+123	123	正号
-123	-123	负号
7 + 4	11	加法
7-4	3	减法
7 * 4	28	乘法
7 // 4	1	整除
7 % 4	3	取余
7 ** 4	2401	乘幂
7 // 0	运行时错误	整除,除数不能为 0
3 * 4 - 3	9	* 的优先级比 - 的优先级高
3 + 4 // 3	4	//的优先级比+的优先级高
3 - 4 - 2	-3	左结合运算
2 ** 2 ** 3	256	右结合运算
pow(2,10)	1024	乘幂(调用数学模块函数)

【例 4.4】　整数运算示例(int_ops.py)。

```
import sys
a = int(sys.argv[1])
b = int(sys.argv[2])
sum = a + b
print(a, ' + ', b, ' = ', sum)
```

程序运行结果如图 4-1 所示。

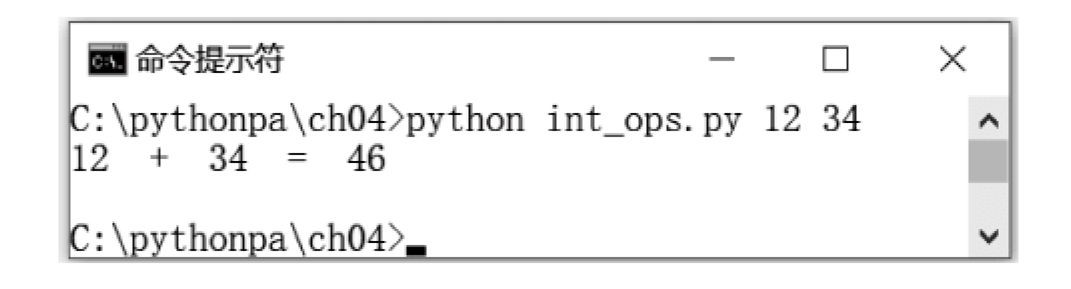


图 4-1　整数运算示例程序运行结果

4.3　float 类型

浮点类型(float)是表示实数的数据类型,与其他计算机语言的双精度(double)和单精度对应。Python 浮点类型的精度与系统相关。

4.3.1　浮点类型字面量

浮点类型字面量可以为带小数点的数字字符串，或用科学记数法表示的数字字符串(前面可以带负号"—")，即浮点型字面量。Python 解释器自动创建 float 型对象实例。

浮点类型字面量的示例如表 4-3 所示。

表 4-3　浮点类型字面量的示例

举　例	说　明
1.23、—24.5、1.0、0.2	带小数点的数字字符串
1.、.2	小数点前后的 0 可以省略
3.14e—10、4E210、4.0e+210	科学记数法(e 或 E 表示底数 10)，例如 $3.14e-10=3.14*10^{-10}$

【例 4.5】　浮点类型字面量示例。

```
>>> 3.14                                    #输出:3.14
>>> type(3.14)                              #输出:<class 'float'>
```

4.3.2　float 对象

float 是 Python 的内置数据类型，用户可以创建 float 类型的对象实例，其基本形式如下。

```
float(x)
```

通过创建 float 对象可以把数值或任何符合格式的字符串转换为 float 对象。

注意： 如果对象 x 不能转换为 float 对象，将导致 TypeError；如果对象 x 转换失败，将导致 ValueError。特殊字符串'Infinity'、'—Infinity'和'NaN'分别用于表示正无穷大、负无穷大和非数值。

【例 4.6】　float 对象示例。

```
>>> float                                    #输出:<class 'float'>
>>> float(123), float('3.14')                #输出:(123.0, 3.14)
>>> float('Infinity'), float('-Infinity'), float('NaN')  #输出:(inf, -inf, nan)
>>> float('123abc')                          #报错.ValueError: could not convert string
                                             #to float: '123abc'
```

4.3.3　float 对象的方法

float 对象包含的主要方法如表 4-4 所示。

表 4-4　float 对象的主要方法

方　法	说　明	示　例
as_integer_ratio()	转换为分数	1.25.as_integer_ratio()　#结果：(5，4) float.as_integer_ratio(1.25)　#结果：(5，4)
hex()	转换为十六进制字符串	12.3.hex()　#结果：'0x1.899999999999ap+3' float.hex(12.3)　#结果：'0x1.899999999999ap+3'
fromhex(string)	十六进制字符串转换为浮点数	float.fromhex('0xFF')　#结果：255.0
is_integer()	判断是否为 int 类型	3.14.is_integer()　#结果：False float.is_integer(2.0)　#结果：True

4.3.4 浮点数的运算

浮点数对象支持关系运算、算术运算、位运算符、内置函数、math 模块中的数学运算函数以及 float 对象方法(参见 4.3.3 节)等运算操作。

在 Python 语言中,常用的 float 数据类型对象的运算表达式如表 4-5 所示。

表 4-5 Python 常用的浮点数运算表达式

表 达 式	结 果	说 明
3.14	3.14	浮点数字面值
6.67e−11	6.67e−11	浮点数字面值
3.14+2.0	5.140000000000001	加法
3.14−2.0	1.1400000000000001	减法
3.14 * 2.0	6.28	乘法
3.14/2.0	1.57	除法
4.0/3.0	1.3333333333333333	除法
3.14 ** 2.0	9.8596	乘幂
2.0/0.0	运行时错误	除法。除数不能为 0
20.0 ** 1000.0	运行时错误	结果太大无法表示
math.sqrt(2.0)	1.4142135623730951	平方根(调用数学模块函数)
math.sqrt(−2.0)	运行时错误	负数的平方根

【例 4.7】 浮点数运算示例(float_ops.py)。

```
import sys
a = float(sys.argv[1])
b = float(sys.argv[2])
c = a * b
print(a, ' * ', b, ' = ', c)
```

程序运行结果如图 4-2 所示。

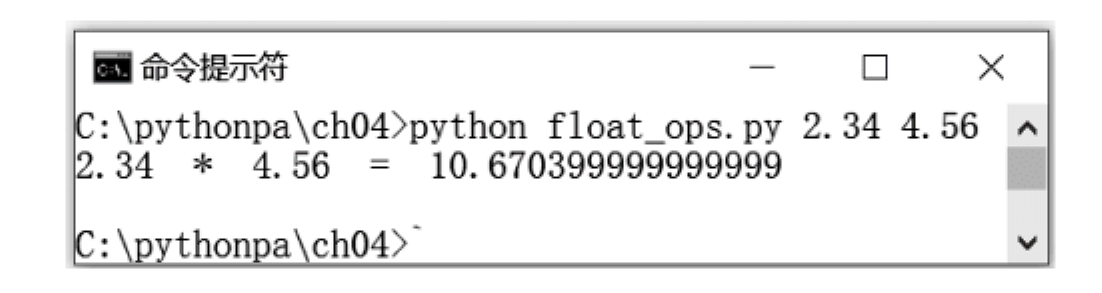


图 4-2 浮点数运算示例程序运行结果

注意:浮点数运算会产生误差。

4.4 complex 类型

4.4.1 复数类型字面量

当数值字符串中包含虚部(j 或 J)时即复数字面量。Python 解释器自动创建 complex 型对象实例。

【例 4.8】 复数字面量示例。

```
>>> 1 + 2j                                    #输出:(1 + 2j)
>>> type(1 + 2j)                              #输出:<class 'complex'>
```

4.4.2　complex 对象

complex 是 Python 的内置数据类型，用户可以创建 complex 类型的对象实例，其基本形式如下。

```
complex(real[, imag])                        # 创建 complex 对象(虚部可选)
```

【例 4.9】　complex 对象示例。

```
>>> complex                                  # 输出:<class 'complex'>
>>> c = complex(4, 5)
>>> c                                        # 输出:(4+5j)
```

4.4.3　complex 对象的属性和方法

complex 对象包含的属性和方法如表 4-6 所示。

表 4-6　complex 对象的属性和方法

属性/方法	说　明	示　例	
real	复数的实部	>>> (1+2j).real	#结果:1.0
imag	复数的虚部	>>> (1+2j).imag	#结果:2.0
conjugate()	共轭复数	>>> (1+2j).conjugate()	#结果:(1-2j)

在 Python 内部复数使用正交笛卡儿坐标表示，所以 z == z.real + z.imag * 1j。

4.4.4　复数的运算

复数对象支持算术运算、cmath 模块中的数学运算函数、complex 对象方法(参见 4.4.3 节)等运算操作。

在 Python 语言中，常用的 complex 数据类型对象的运算表达式如表 4-7 所示。

表 4-7　Python 常用的复数运算表达式

表　达　式	结　果	说　明
1+2j	(1+2j)	复数字面量
(1+2j) + (3+4j)	(4+6j)	加法
(1+2j) − (3+4j)	(−2−2j)	减法
(1+2j) * (3+4j)	(−5+10j)	乘法
(1+2j) / (3+4j)	(0.44+0.08j)	除法
(1+2j) ** 2.0	(−3+4j)	乘幂
(1+2j) / 0.0	运行时错误	除法。除数不能为 0
cmath.sqrt(1+2j)	(1.272019649514069+0.7861513777574233j)	平方根(调用数学模块函数)
cmath.sqrt(−2.0)	1.4142135623730951j	复数的平方根

【例 4.10】　复数运算示例。

```
>>> a = 1 + 2j
>>> b = complex(4, 5)                        # 复数 4 + 5j
>>> a + b                                    # 复数相加.输出:(5+7j)
>>> import cmath
>>> cmath.sqrt(b)                            # 复数的平方根
(2.280693341665298+1.096157889501519j)
```

4.5 bool 类型

Python 的 bool 数据类型用于逻辑运算。

4.5.1 布尔值字面量

bool 数据类型包含两个值：True(真)或 False(假)。

【例 4.11】 布尔值字面量示例。

```
>>> True,False                          #输出:(True, False)
>>> type(True),type(False)              #输出:(<class 'bool'>, <class 'bool'>)
```

4.5.2 bool 对象

用户可以创建 bool 类型的对象实例,其基本形式如下。

```
bool(x)
```

通过创建 bool 对象可以把数值或任何符合格式的字符串或其他对象转换为 bool 对象。

【例 4.12】 bool 对象示例。

```
>>> bool(0)                             #输出:False
>>> bool(1)                             #输出:True
>>> bool("abc")                         #输出:True
```

4.5.3 逻辑运算符

逻辑运算符即布尔运算符,用于检测两个以上条件的情况,即多个 bool 值的逻辑运算,其结果为 bool 类型值。

逻辑运算符除逻辑非(not)是一元运算符以外,其余均为二元运算符。逻辑运算符用于将操作数进行逻辑运算,结果为 True 或 False。表 4-8 按优先级从高到低的顺序列出了 Python 中的逻辑运算符。

表 4-8 Python 中的逻辑运算符

运算符	含义	说明	优先级	实例	结果
not	逻辑非	当操作数为 False 时返回 True;当操作数为 True 时返回 False	1	not True	False
				not False	True
and	逻辑与	当两个操作数均为 True 时结果才为 True,否则为 False	2	True and True	True
				True and False	False
				False and True	False
				False and False	False
or	逻辑或	当两个操作数中有一个为 True 时结果即为 True,否则为 False	3	True or True	True
				True or False	True
				False or True	True
				False or False	False

注意：

(1) Python 中的任意表达式都可以被评价为布尔逻辑值,故均可以参与逻辑运算。

例如：

```
>>> not 0                                        #输出:True
>>> not 'a'                                      #输出:False
```

(2) C = A or B。如果 A 不为 0 或者不为空或者为 True，返回 A；否则返回 B。通常仅在必要时计算第二个操作数，即如果 A 不为 0 或者不为空或者为 True，则不用计算 B，也就是"短路"计算。例如：

```
>>> 1 or 2                                       #输出:1
>>> 0 or 2                                       #输出:2
>>> False or True                                #输出:True
>>> True or False                                #输出:True
```

(3) C = A and B。如果 A 为 0 或者为空或者为 False，返回 A；否则返回 B。通常仅在必要时计算第二个操作数，即如果 A 为 0 或者为空或者为 False，则不用计算 B，也就是"短路"计算。例如：

```
>>> 1 and 2                                      #输出:2
>>> 0 and 2                                      #输出:0
>>> False and 2                                  #输出:False
>>> True and 2                                   #输出:2
```

这种写法常用于不确定 A 是否为空值时把 B 作为候补来赋值给 C。

4.6 str 类型

字符串(str)是一个有序的字符集合。在 Python 中没有独立的字符数据类型，字符即长度为 1 的字符串。

Python 的内置数据类型 str 用于字符串处理。str 对象的值为字符系列。str 对象(字符串)是不可变对象。

4.6.1 字符串字面量

使用单引号或双引号括起来的内容是字符串字面量，Python 解释器自动创建 str 型对象实例。Python 字符串字面量可以使用以下 4 种方式定义。

(1) 单引号(' ')：包含在单引号中的字符串，其中可以包含双引号。

(2) 双引号(" ")：包含在双引号中的字符串，其中可以包含单引号。

(3) 三单引号(''' ''')：包含在三单引号中的字符串，可以跨行。

(4) 三双引号(""" """)：包含在三双引号中的字符串，可以跨行。

【例 4.13】 字符串字面量示例。

```
>>> 'abc'                                        #输出:'abc'
>>> "Hello"                                      #输出:'Hello'
>>> type("python")                               #输出:<class 'str'>
```

注意：两个紧邻的字符串，如果中间只有空格分隔，则自动拼接为一个字符串。例如：

```
>>> 'Blue' 'Sky'                                 #输出:'BlueSky'
```

4.6.2 字符串编码

Python 3 中的字符默认为 16 位 Unicode 编码，ASCII 码是 Unicode 编码的子集。例如，字符'A'的 ASCII 码为 65，对应的八进制为 101，对应的十六进制为 41。

使用 u''或 U''的字符串称为 Unicode 字符串。在 Python 3 中默认为 Unicode 字符串。

```
>>> u'abc'                                  # 输出:'abc'
```

使用内置函数 ord()可以把字符转换为对应的 Unicode 码；使用内置函数 chr()可以把十进制数转换为对应的字符。例如：

```
>>> ord('A')                                # 输出:65
>>> chr(65)                                 # 输出:'A'
>>> ord('张')                               # 输出:24352
>>> chr(24352)                              # 输出:'张'
```

4.6.3　转义字符

特殊符号(不可打印字符)可以使用转义序列表示。转义序列以反斜杠开始,紧跟一个字母,例如"\n"(新行)和"\t"(制表符)。如果希望字符串中包含反斜杠,则它前面必须还有另一个反斜杠。

Python 转义字符如表 4-9 所示。

表 4-9　特殊符号的转义序列

转义序列	字　符	转义序列	字　符
\'	单引号	\n	换行(LF)
\"	双引号	\r	回车(CR)
\\	反斜杠	\t	水平制表符(HT)
\a	响铃(BEL)	\v	垂直制表符(VT)
\b	退格(BS)	\ooo	八进制 Unicode 码对应的字符
\f	换页(FF)	\xhh	十六进制 Unicode 码对应的字符

【例 4.14】　转义字符示例。

```
>>> s = 'a\tb\tc\\td'
>>> s                                       # 输出:'a\tb\tc\\td'
>>> print(s)                                # 输出: a       b       c\td
```

转义字符后跟 Unicode 编码也可以表示字符。例如：

```
>>> '\101'                                  # 输出:'A'
>>> '\x41'                                  # 输出:'A'
```

使用 r''或 R''的字符串称为原始字符串,其中包含的任何字符都不进行转义。

```
>>> s = r'换\t 行\t 符\n'
>>> s                                       # 输出:'换\\t 行\\t 符\\n'
>>> print(s)                                # 输出:换\t 行\t 符\n
```

4.6.4　str 对象

str 是 Python 的内置数据类型,创建 str 类型的对象实例的基本形式如下。

```
str(object = '')                            # 创建 str 对象,默认为空字符串
```

通过创建 str 对象可以把任意对象转换为 str 对象,返回 object.__str__(),如果对象没有定义__str__(),则返回 repr(object)。

【例 4.15】 str 对象示例。

```
>>> str(123)                                  #输出:'123'
>>> str(True)                                 #输出:'True'
>>> str(3.14)                                 #输出:'3.14'
```

4.6.5 str 对象的属性和方法

使用 str 对象提供的方法可以实现常用的字符串处理功能。str 对象是不可变对象,故调用方法返回的字符串是新创建的对象。str 对象的方法有两种调用方式,即字符串对象的方法和 str 类方法。

【例 4.16】 str 对象方法示例。

```
>>> s = 'abc'
>>> s.upper()                     #字符串对象 s 的方法.输出:'ABC'
>>> str.upper(s)                  #str 类方法,字符串 s 作为参数.输出:'ABC'
```

4.6.6 字符串的运算

字符串对象支持关系运算、使用运算符"+"拼接两个字符串、内置函数、str 对象方法等运算操作。

字符串实际上是字符序列,故支持序列数据类型的基本操作,包括索引访问、切片操作、连接操作、重复操作、成员关系操作,以及求字符串长度、最大值、最小值等。例如,通过 len(s)可以获取字符串 s 的长度;如果其长度为 0,则为空字符串。具体内容可以参见第 5 章。

在 Python 语言中,常用的 str 数据类型对象的运算表达式如表 4-10 所示。

表 4-10 Python 常用的字符串表达式

表　达　式	结　　果	说　　明
'Hello, ' + 'World'	'Hello, World'	字符串拼接
'123' + '456'	'123456'	字符串拼接(不是两数相加)
'1234' + ' + ' + '99'	'1234 + 99'	两次字符串拼接
'123' + 456	运行时错误	第二个操作数不是 str 数据类型

4.6.7 对象转换为字符串

使用内置函数 str()可以把数值转换为字符串。实际上,在使用 print(123)输出数值时将自动调用 str(123)函数把 123 转换为字符串,然后输出。

Python 还提供了另一个内置函数 repr(),该函数返回一个对象的更精确的字符串表示形式。

在大多数情况下,内置函数 repr()和 str()的结果一致。

【例 4.17】 对象转换为字符串示例。

```
>>> c = 1/3
>>> str(c)                                    #输出:'0.3333333333333333'
>>> repr(c)                                   #输出:'0.3333333333333333'
```

4.6.8 字符串的格式化

通过字符串的格式化可以输出特定格式的字符串。Python 中字符串的格式化有以下几

种方式：

- **字符串.format(值 1, 值 2, …)**
- **str.format(格式字符串 1,值 1, 值 2, …)**
- **format(值, 格式字符串)**
- **格式字符串 % (值 1, 值 2, …)** #兼容 Python 2 的格式,不建议使用

有关字符串格式化的详细信息请参见本书中的第 5.5.3 节。

例如：

```
>>> "学生人数{0},平均成绩{1}".format(15, 81.2)
'学生人数 15,平均成绩 81.2'
>>> str.format("学生人数{0},平均成绩{1:2.2f}", 15, 81.2)
'学生人数 15,平均成绩 81.20'
>>> format(81.2, "0.5f")                              #输出:'81.20000'
>>> "学生人数 %4d,平均成绩 %2.1f" % (15, 81)
'学生人数 15,平均成绩 81.0'
```

【例 4.18】 字符串示例(string.py)：格式化输出字符串堆积的三角形。其中,str.center()方法用于字符串两边填充；str.rjust(width[, fillchar])方法用于字符串右填充,具体可以参见本书中的第 15.2 节。

```
print("1".center(20))                           #一行 20 个字符,居中对齐
print(format("121", "^20"))                     #一行 20 个字符,居中对齐
print(format("12321", "^20"))                   #一行 20 个字符,居中对齐
print("1".rjust(20,"*"))                        #一行 20 个字符,右对齐,加*号
print(format("121", "*>20"))                    #一行 20 个字符,右对齐,加*号
print(format("12321", "*>20"))                  #一行 20 个字符,右对齐,加*号
```

程序运行结果如下。

```
         1
        121
       12321
*******************1
*****************121
***************12321
```

4.6.9 格式化字符串变量

在 Python 3.6 中增加了对格式化字符串变量的支持,以 f 开始的字符串可以包含嵌入在花括号“{}”中的变量,称之为字符串变量替换(插值)。例如：

```
>>> name = "Fred"
>>> f"He said his name is {name}."              #输出:'He said his name is Fred.'
>>> score, width, precision = 12.34567, 10, 4
>>> f"result: {score:{width}.{precision}}"      #输出:'result:      12.35'
```

4.7 比较关系运算和条件表达式

4.7.1 条件表达式

条件表达式通常用在选择语句中,用于判断是否满足某种条件。最简单的条件表达式可

以是一个常量或变量，复杂的条件表达式包含关系比较运算符和逻辑运算符。条件表达式的最后评价为 bool 值 True(真)或 False(假)。

Python 的评价方法如下：如果表达式的结果为数值类型(0)、空字符串("")、空元组(())、空列表([])、空字典({})，则其 bool 值为 False(假)；否则其 bool 值为 True(真)。例如，123、"abc"、(1,2)均为 True。

【例 4.19】　条件表达式示例。

```
>>> bool(123),bool("abc"),bool((1,2)),bool([0]),bool(0)
(True, True, True, True, False)
>>> bool(1 > 2),bool(1 > 2 or 3 > 2),bool(1 <= 2 and 3 > 2)
(False, True, True)
```

4.7.2　关系和测试运算符

关系和测试运算符是二元运算符。关系运算符用于对两个操作数的大小进行比较。若关系成立，则比较的结果为 True，否则为 False。

原则上，关系比较运算符应该是两个相同类型的对象之间的比较。例如：

```
>>> 1 > 2                      #输出:False
>>> "ab123" > "ab12"           #输出:True
```

不同类型的对象也允许进行比较，但会导致错误。数值类型(包括布尔型，True 自动转换为 1，False 自动转换为 0)之间可以进行比较。例如：

```
>>> 1 > 1.23                   #输出:False
>>> 2 > True                   #输出:True
>>> 123 >"abc"                 #报错.TypeError: unorderable types: int() > str()
```

Python 语言的关系和测试运算符如表 4-11 所示。

表 4-11　关系和测试运算符

运算符	表达式	含义	实例	结果
==	x== y	x 等于 y	"ABCDEF" == "ABCD"	False
!=	x!= y	x 不等于 y	"ABCD" != "abcd"	True
>	x> y	x 大于 y	"ABC" > "ABD"	False
>=	x>= y	x 大于等于 y	123 >= 23	True
<	x< y	x 小于 y	"ABC" < "上海"	True
<=	x<= y	x 小于等于 y	"123" <= "23"	True
is	x is y	x 和 y 是同一个对象	x=y=1; x is y	True
			x=1; y=2; x is y	False
is not	x is not y	x 和 y 不是同一个对象	x=1; y=2; x is not y	True
in	x in y	x 是 y 的成员(y 是容器，例如元组)	1 in (1, 2, 3)	True
			"A" in "ABCDEF"	True
not in	x not in y	x 不是 y 的成员(y 是容器，例如元组)	1 not in (1, 2, 3)	False

注意：

(1) 关系运算符的优先级相同。

(2) 对于两个预定义的数值类型，关系运算符按照操作数的数值大小进行比较。

(3) 对于字符串类型，关系运算符比较字符串的值，即按字符的 ASCII 码值从左到右一一比较：首先比较两个字符串的第一个字符，其 ASCII 码值大的字符串大，若第一个字符相等，则继续比较第二个字符，依此类推，直到出现不同的字符为止。

4.8 算术运算符和位运算符

4.8.1 算术运算符

Python 提供了丰富的算术运算符，用于进行包括四则运算的各种算术运算。表 4-12 以优先级顺序列出了 Python 中的算术运算符。假设该表中的 n 为整型变量，取值为 8。

表 4-12 算术运算符

运算符	含 义	说 明	优先级	实 例	结 果
**	乘幂	操作数的乘幂	1	n ** 3	512
+	一元+	操作数的值	2	+n	8
-	一元-	操作数的反数	2	-n	-8
*	乘法	操作数的积	3	n * n * 2	128
/	除法	第二个操作数除第一个操作数	3	10 / n	1.25
//	整数除法	两个整数相除，结果为整数	3	10 // n	1
%	模数	第二个操作数除第一个操作数后的余数	3	10 % n	2
+	加法	两个操作数之和	4	10 + n	18
-	减法	从第一个操作数中减去第二个操作数	4	n - 10	-2

4.8.2 位运算符

位运算符用于按二进制位进行逻辑运算，操作数必须为整数。Python 中的位运算符如表 4-13 所示。

表 4-13 位运算符

运 算 符	用 法	含 义	优 先 级	实 例	结 果
~	~op	按位求补	1	~0x1	-2(-0x2)
<<	op1<<op2	将 op1 左移 op2 位	2	0xf0<<4	3840(0xf00)
>>	op1>>op2	将 op1 右移 op2 位	2	0xf0>>4	15(0xf)
&	op1&op2	按位逻辑与	3	0xff00 & 0xf0f0	61440(0xf000)
^	op1 ^op2	按位逻辑异或	4	0xff00 ^ 0xf0f0	4080(0xff0)
|	op1|op2	按位逻辑或	5	0xff00 | 0xf0f0	65520(0xfff0)

【例 4.20】 位运算符示例(op_bit.py)。

```
print("~0x1 结果为:", hex(~0x1))
print("0b1110000 << 4 结果为:", bin(0b111110000 << 4))
print("0b1110000 >> 4 结果为:", bin(0b111110000 >> 4))
print("0b1111111100000000 & 0b1111000011110000 结果为:", bin(0b1111111100000000 & 0b1111000011110000))
```

```
print("0b1111111100000000 | 0b1111000011110000 结果为:", bin(0b1111111100000000 | 0b1111000011110000))
print("0b1111111100000000 ^ 0b1111000011110000 结果为:", bin(0b1111111100000000 ^ 0b1111000011110000))
```

程序运行结果如下。

```
~0x1 结果为: -0x2
0b11110000 << 4 结果为: 0b111100000000
0b11110000 >> 4 结果为: 0b1111
0b1111111100000000 & 0b1111000011110000 结果为: 0b1111000000000000
0b1111111100000000 | 0b1111000011110000 结果为: 0b1111111111110000
0b1111111100000000 ^ 0b1111000011110000 结果为: 0b111111110000
```

4.9　混合运算和数值类型转换

4.9.1　隐式转换

int、float 和 complex 对象可以进行混合运算。如果表达式中包含 complex 对象，则其他对象自动转换（隐式转换）为 complex 对象，结果为 complex 对象；如果表达式中包含 float 对象，则其他对象自动转换（隐式转换）为 float 对象，结果为 float 对象。

【例 4.21】　隐式类型转换示例。

```
>>> f = 123 + 1.23
>>> f                              #输出:124.23
>>> type(f)                        #输出:<class 'float'>
>>> 123 + True                     #True 转换为 1.输出:124
>>> 123 + False                    # False 转换为 0.输出:123
```

注意，在混合运算中 True 自动转换为 1，False 自动转换为 0。

4.9.2　显式转换

"显式转换"又称为"强制转换"，使用 target-type(value)将表达式强制转换为所需的数据类型。如果未定义相应的转换运算符，则强制转换会失败。显式转换实际上使用目标类型的构造函数创建其对象。

int(x)、float(x)、bool(x)、str(x)分别把对象转换为整数、浮点数、布尔值和字符串。

【例 4.22】　显式类型转换示例。

```
>>> int(1.23)                #输出:1
>>> float(10)                #输出:10.0
>>> bool("abc")              #输出:True
>>> float("123xyz")          #报错.ValueError: could not convert string to float: '123xyz
'
```

显式数值转换可能导致精度损失，也可能引发异常（例如 OverflowError）。例如：

```
>>> i = 9999 * * 9999
>>> float(i)                 #报错.OverflowError: long int too large to convert to float
```

【例 4.23】　数值数据类型示例(profit.py)：计算复利。

```
nb = float(input("请输入本金:"))         #输入本金并转换为浮点数
nr = float(input("请输入年利率:"))       #输入年利率并转换为浮点数
```

```
ny = int(input("请输入年数:"))            #输入年数并转换为整数
amount = nb * (1 + nr/100) ** ny         #计算复利
print("本金利率和为:%0.2f" % amount)      #输出复利,保留两位小数
```

程序运行结果如下。

```
请输入本金:1000
请输入年利率:6
请输入年数:10
本金利率和为: 1790.85
```

4.10 内置标准数学函数

4.10.1 内置数学运算函数

在 Python 中包含了若干用于数学运算的内置函数,如表 4-14 所示。

表 4-14 用于数学运算的内置函数

函数	含义	实例	结果
abs(x)	数值 x 的绝对值。如果 x 为复数,则返回 x 的模	abs(−1.2)	1.2
		abs(1−2j)	2.23606797749979
divmod(a,b)	返回 a 除以 b 的商和余数	divmod(5,3)	(1, 2)
pow(x, y[, z])	返回 x 的 y 次幂(x ** y)。如果指定 z,则为 pow(x, y) % z	pow(2,10)	1024
		pow(2,10,10)	4
round(number[, ndigits])	四舍五入取整。如果指定 ndigits,则保留 ndigits 小数	round(3.14159)	3
		round(3.14159,4)	3.1416
sum(iterable[, start])	求和	sum((1, 2, 3))	6
		sum((1, 2, 3), 44)	50

4.10.2 数制转换函数

Python 包含如表 4-15 所示的内置函数,用于不同数制之间的转换。

表 4-15 数制转换函数

函数	说明	示例
bin(number)	数值转换为二进制字符串	bin(100) #结果:'0b1100100'
hex(number)	数值转换为十六进制字符串	hex(100) #结果:'0x64'
oct(number)	数值转换为八进制字符串	oct(100) #结果:'0o144'

4.11 复 习 题

一、选择题

1. 在下列数据类型中,Python 不支持的是________。

   A. char　　B. int　　C. float　　D. list

2. Python 语句 print(type(1J))的输出结果是________。

A. < class 'complex'>　　B. < class 'int'>
C. < class 'float'>　　D. < class 'dict'>

3. Python 语句 print(type(1/2))的输出结果是________。

A. < class 'int'>　　B. < class 'number'>
C. < class 'float'>　　D. < class 'double'>

4. Python 语句 print(type(1//2))的输出结果是________。

A. < class 'int'>　　B. < class 'number'>
C. < class 'float'>　　D. < class 'double'>

5. Python 语句序列“a=121+1.21; print(type(a))”的输出结果是________。

A. < class 'int'>　　B. < class 'float'>　　C. < class 'double'>　　D. < class 'long'>

6. Python 语句 print(0xA + 0xB)的输出结果是________。

A. 0xA + 0xB　　B. A + B　　C. 0xA0xB　　D. 21

7. Python 语句序列“x='car'; y=2; print(x+y)”的输出结果是________。

A. 语法错　　B. 2　　C. 'car2'　　D. 'carcar'

8. Python 表达式 sqrt(4) * sqrt(9)的值为________。

A. 36.0　　B. 1296.0　　C. 13.0　　D. 6.0

9. 关于 Python 中的复数,下列说法错误的是________。

A. 表示复数的语法是 real+image j　　B. 实部和虚部都是浮点数
C. 虚部必须后缀为 j,且必须是小写　　D. 方法 conjugate()返回复数的共轭复数

10. Python 语句 print(chr(65))的运行结果是________。

A. 65　　B. 6　　C. 5　　D. A

11. 关于 Python 字符串,下列说法错误的是________。

A. 字符即长度为 1 的字符串
B. 字符串以\0 标识字符串的结束
C. 用户既可以用单引号,也可以用双引号创建字符串
D. 在三引号字符串中可以包含换行回车等特殊字符

二、填空题

1. Python 中内置的 4 种数值类型为________。
2. Python 内置的序列数据类型包括________。
3. Python 表达式 10 + 5 // 3-True+False 的值为________。
4. Python 表达式 3 ** 2 ** 3 的值为________。
5. Python 表达式 17.0 / 3 ** 2 的值为________。
6. Python 表达式 0 and 1 or not 2 < True 的值为________。
7. Python 语句 print(pow(-3, 2), round(18.67, 1), round(18.67, -1))的输出结果是________。
8. Python 语句 print(round(123.84,0), round(123.84,-2), floor(15.5))的输出结果是________。
9. Python 语句 print(int('20', 16), int('101', 2))的输出结果是________。
10. Python 语句 print(hex(16), bin(10))的输出结果是________。
11. Python 语句 print(2.5.as_integer_ratio())的输出结果是________。

12. Python 语句 print(float.as_integer_ratio(1.5))的输出结果是________。

13. Python 语句 print(gcd(12, 16),divmod(7,3))的输出结果是________。

14. Python 语句 print((2-3j).conjugate() * complex(2, 3))的输出结果是________。

15. Python 语句 print(sum((1, 2, 3)), sum((1, 2, 3), 10))的输出结果是________。

16. Python 语句 print(abs(-3.2), abs(1-2j))的输出结果是________。

17. Python 的标准随机数生成器模块是________。

18. 数学表达式 $\sin 15° + \dfrac{e^x - 5x}{\sqrt{x^2+1}} - \ln(3x)$ 的 Python 表达式为________。

19. 数学表达式 $\dfrac{\dfrac{\frac{1}{2}cd}{c+d} - \dfrac{4\pi}{c-d}}{a+b}$ 的 Python 表达式为________。

20. Python 的内置字典数据类型为________。

21. Python 语句序列"x=True; y=False; z=False; print(x or y and z);"的运行结果是________。

22. Python 语句序列"x=0; y=True; print(x>=y and 'A'<'B');"的运行结果是________。

23. 在直角坐标系中,(x,y)是坐标系中任意点的位置,用 x 和 y 表示第一象限或者第二象限的 Python 表达式为________。

24. 判断整数 i 能否同时被 3 和 5 整除的 Python 表达式为________。

25. 已知"a=3; b=5; c=6; d=True",则表达式 not d or a >=0 and a+c>b+3 的值是________。

26. Python 表达式 16-2 * 5>7 * 8/2 or "XYZ"!="xyz" and not(10-6>18/2)的值为________。

27. 执行下列 Python 语句将产生的结果是________。

```
m = True;n = False;p = True
b1 = m | n ^ p; b2 = n | m ^ p
print(b1,b2)
```

28. Python 语句 print(chr(ord('B')))的执行结果是________。

29. Python 语句 print("hello" 'world')的执行结果是________。

三、思考题

1. Python 包括哪 4 种内置的数值类型?
2. Python 包括哪些不可变序列数据类型? 哪些可变序列数据类型?
3. Python 字符串字面量有哪 4 种定义方式?
4. Python 有哪几种类型转换方式? 各方式是如何进行类型转换的?
5. 阅读下面的 Python 程序,请问输出结果是什么?

```
from decimal import *
ctx = getcontext(); ctx.prec = 2;print(Decimal('1.78'))
print(Decimal('1.78') + 0); ctx.rounding = ROUND_UP
print(Decimal('1.65') + 0); print(Decimal('1.62') + 0); print(Decimal('-1.45') + 0)
print(Decimal('-1.42') + 0); ctx.rounding = ROUND_HALF_UP
print(Decimal('1.65') + 0);print(Decimal('1.62') + 0); print(Decimal('-1.45') + 0)
```

```
ctx.rounding = ROUND_HALF_DOWN; print(Decimal('1.65') + 0);print(Decimal('-1.45') + 0)
```

6. 阅读下面的 Python 语句，请问输出结果是什么？程序的功能是什么？

```
import random
a = random.randint(100,999)          #随机产生一个 3 位整数
b = (a % 10) * 100 + (a // 10 % 10) * 10 + a // 100
print("原数 = ", a,",变换后 = ", b)
```

7. 下列 Python 语句的输出结果是________。

```
print("数量{0},单价{1}".format(100, 285.6))
print(str.format("数量{0},单价{1:3.2f}", 100, 285.6))
print("数量 %4d,单价 %3.3f" % (100, 285.6))
```

8. 下列 Python 语句的输出结果是________。

```
print("1".rjust(20," "))
print(format("121", " >20"))
print(format("12321", " >20"))
```

9. 下列 Python 语句的运行结果为________。

```
x = False; y = True; z = False
if x or y and z: print("yes")
else: print("no")
```

10. 下列 Python 语句的运行结果为________。

```
x = True; y = False; z = True;
if not x or y: print(1)
elif not x or not y and z: print(2)
elif not x or y or not y and x: print(3)
else: print(4)
```

11. 阅读下面的 Python 程序，请问输出结果是什么？

```
print(1 or 2, 0 or 2, False or True,True or False,False or 2,sep = ' ')
print(1 and 2, 0 and 2, False and 2,True and 2,False and True,sep = ' ')
```

12. 阅读下面的 Python 程序，请问输出结果是什么？

```
print("T",end = ' ') if not 0 else print('F',end = ' ')
print("T",end = ' ') if 6 else print('F',end = ' ')
print("T",end = ' ') if "" else print('F',end = ' ')
print("T",end = ' ') if "abc" else print('F',end = ' ')
print("T",end = ' ') if () else print('F',end = ' ')
print("T",end = ' ') if (1,2) else print('F',end = ' ')
print("T",end = ' ') if [] else print('F',end = ' ')
print("T",end = ' ') if [1,2] else print('F',end = ' ')
print("T",end = ' ') if {} else print('F',end = ' ')
print("T",end = ' ') if {1,2} else print('F',end = ' ')
```

4.12 上机实践

1. 完成本书中的例 4.1～例 4.23，熟悉 Python 语言中常用内置数据类型的运算和操作。

2. 编写程序，格式化输出杨辉三角。杨辉三角即二项式定理的系数表，各元素满足如下条件：第一列及对角线上的元素均为 1；其余每个元素等于它上一行同一列元素与前一列元素之和。运行效果参见图 4-3 所示。

提示：

用户可以使用“print("1".center(20))”的语句形式在一行上打印 20 个字符，并且居中对齐。

3. 输入直角三角形的两个直角边，求三角形的周长和面积，以及两个锐角的度数。结果均保留一位小数。其运行效果如图 4-4 所示。

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

图 4-3 杨辉三角运行效果

```
请输入直角三角形的直角边A(>0):    4
请输入直角三角形的直角边B(>0):    4
直角三角形三边分别为:  a=4.0, b=4.0, c=5.7
三角形的周长 = 13.7, 面积 = 8.0
三角形两个锐角的度数分别为:  45.0 和 45.0
```

图 4-4 直角三角形运行效果

提示：

（1）math.asin()函数返回正弦值为指定数字的弧度；math.acos()函数返回余弦值为指定数字的弧度。

（2）将弧度转换为角度的公式为角度 $=\dfrac{\text{弧度} * 180}{\pi}$。

（3）可以使用“round(asin(sinA) * 180 / pi, 0)”的语句形式求锐角 A 的度数。

（4）可以使用“print(str.format("三角形的周长 = {0: 1.1f}，面积 = {1: 1.1f}", p, area))”的语句形式按题目要求输出三角形的周长和面积。

4. 编程产生 0～100（包含 0 和 100）的 3 个随机数 a、b 和 c，要求至少使用两种不同的方法，将 3 个数按从小到大的顺序排序。其运行效果如图 4-5 所示（其中，a、b 和 c 的值随机生成）。

提示：

（1）方法一：先比较 a 和 b，使得 a<b；然后比较 a 和 c，使得 a<c，此时 a 最小；最后比较 b 和 c，使得 b<c。

（2）方法二：利用 max()函数和 min()函数求 a、b、c 中的最大数、最小数，而 3 个数之和减去最大数和最小数就是中间数。

（3）利用 random.randint(0,100)生成 0～100（包含 0 和 100）的随机数。

（4）利用 max(a, b, c)返回 a、b 和 c 的最大值；利用 min(a, b, c)返回 a、b 和 c 的最小值。

5. 编程计算有固定工资收入的党员每月所交纳的党费。工资基数 3000 元及以下者，交纳工资基数的 0.5%；工资基数 3000～5000 元者，交纳工资基数的 1%；工资基数在 5000～10000 元者，交纳工资基数的 1.5%；工资基数超过 10000 元者，交纳工资基数的 2%。运行效果如图 4-6 所示。

$$\text{党费 } f=\begin{cases}0.5\% * \text{salary} & \text{salary} \leqslant 3000 \\ 1\% * \text{salary} & 3000<\text{salary} \leqslant 5000 \\ 1.5\% * \text{salary} & 5000<\text{salary} \leqslant 10000 \\ 2\% * \text{salary} & \text{salary}>10000\end{cases}$$

```
原始值:      a=97,  b=89,  c=99
(方法一)升序值:   a=89,  b=97,  c=99
(方法二)升序值:   a=89,  b=97,  c=99
```

图 4-5 3 个数比较大小的运行效果

```
请输入有固定工资收入的党员的月工资: 12000
月工资 = 12000, 交纳党费 = 240.0
```

图 4-6 党费交纳运行效果

6. 编程实现袖珍计算器，要求输入两个操作数和一个操作符（＋、－、＊、/、％），根据操作符输出运算结果。注意“/”和“％”运算符的零除异常问题。其运行效果如图 4-7 所示。

```
请输入操作数x: 3
请输入操作数y: 6
请输入操作符: +
3.0 + 6.0 = 9.0
```

```
请输入操作数x: 3
请输入操作数y: 0
请输入操作符: /
分母=0，零除异常!
```

图 4-7　袖珍计算器运行效果

7. 输入三角形的 3 条边 a、b、c，判断此 3 边是否可以构成三角形。若能，进一步判断三角形的性质，即为等边、等腰、直角或其他三角形。本题的判断准则参见表 4-16。其运行效果如图 4-8 所示。

表 4-16　各类三角形的判断准则

形　　状	满 足 条 件
三角形	3 条边均大于零，且任意两边之和大于第三边
等边三角形	3 条边均相等的三角形
等腰三角形	只有两边相等的三角形
直角三角形	勾股定理：$斜边^2 = 直角边\ 1^2 + 直角边\ 2^2$

```
请输入三角形的边a: 1
请输入三角形的边b: 2
请输入三角形的边c: 3
无法构成三角形!
```

```
请输入三角形的边a: 5
请输入三角形的边b: 5
请输入三角形的边c: 5
该三角形为等边三角形!
```

```
请输入三角形的边a: 5
请输入三角形的边b: 5
请输入三角形的边c: 7
该三角形为等腰三角形!
```

```
请输入三角形的边a: 3
请输入三角形的边b: 4
请输入三角形的边c: 5
该三角形为直角三角形!
```

图 4-8　判断三角形运行效果

8. 编程实现鸡兔同笼问题。已知在同一个笼子里共有 h 只鸡和兔，鸡和兔的总脚数为 f，其中 h 和 f 由用户输入，求鸡和兔各有多少只？要求使用两种方法：一是求解方程；二是利用循环进行枚举测试。其运行效果如图 4-9 所示。

```
请输入总头数:  10
请输入总脚数(必须是偶数): 25
请输入总脚数(必须是偶数): 26
方法一: 鸡: 7 只,  兔: 3 只
方法二: 鸡: 7 只,  兔: 3 只
```

(a) 合理解

```
请输入总头数:  10
请输入总脚数(必须是偶数): 10
方法一: 无解，请重新运行测试!
方法二: 无解，请重新运行测试!
```

(b) 无解

图 4-9　鸡兔同笼运行效果

提示：

（1）已知鸡和兔的总头数为 h、总脚数为 f，假设鸡有 c 只、兔有 r 只。

（2）方法一：求解方程法。由公式：

$$\begin{cases} c + r = h \\ 2c + 4r = f \end{cases}$$

解得：

$$\begin{cases} r = \dfrac{f}{2} - h \\ c = h - r \end{cases}$$

由公式推得，鸡和兔的总脚数 f 必须是偶数，并且鸡和兔的只数必须是非负整数。

（3）方法二：利用循环进行枚举测试。鸡的只数 c 的取值范围为 0～h，则兔的数量 r 为（h－c），如果满足条件（2c＋4r＝＝f），则求得解。

9. 输入任意实数 x，计算 e^x 的近似值，直到最后一项的绝对值小于 10^{-6} 为止。其运行效

果如图 4-10 所示。

$$e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

10. 输入任意实数 a(a⩾0)，用迭代法求 $x=\sqrt{a}$，要求计算的相对偏差小于 10^{-6}。其运行效果如图 4-11 所示。求平方根的迭代公式为：

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

```
请输入x: 2
Pow(e,x)  =   7.3890560703259105
```

图 4-10　e^x 运行效果

```
2 的算术平方根= 1.4142135623746899
```

图 4-11　平方根运行效果

11. 我国汉代有位大将，名叫韩信。他每次集合部队，只要求部下先后按 1～3、1～5、1～7 报数，然后再报告一下各队每次报数的余数，他就知道到了多少人。他的这种巧妙算法被人们称为“鬼谷算”，也叫“隔墙算”，或称为“韩信点兵”，外国人还称它为“中国余数定理”。即有一个数，用 3 除余 2，用 5 除余 3，用 7 除余 2，请问 0～1000 中这样的数有哪些？其运行效果如图 4-12 所示。

12. 一球从 100 米的高度自由落下，每次落地后反弹回原高度的一半，再落下。求小球在第 10 次落地时共经过多少米？第 10 次反弹多高？其运行效果如图 4-13 所示。

```
0~1000中用3除余2，用5除余3，用7除余2的数有：
23 128 233 338 443 548 653 758 863 968
```

图 4-12　韩信点兵运行效果

```
小球在第10次落地时，共经过199.80米
第10次反弹0.20米
```

图 4-13　自由落体运行效果

13. 猴子吃桃问题。猴子第一天摘下若干个桃子，当天吃掉一半多一个；第二天接着吃了剩下的桃子的一半多一个；以后每天都吃了前一天剩下的桃子的一半多一个。到第 8 天发现只剩一个桃子了。请问猴子第一天共摘了多少个桃子？其运行效果如图 4-14 所示。

提示：

这是一个递推问题。假设第 n 天的桃子数为 P_n，前一天(第 n−1 天)的桃子数为 P_{n-1}，则 $P_n = \frac{1}{2}P_{n-1} - 1$，即 $P_{n-1} = 2(P_n + 1)$。现在已知第 8 天(n=8)的桃子数 $P_8 = 1$，根据公式得第 7 天的桃子数 $P_7 = 4$，依此类推，可以求得第 1 天的桃子数 P_1。

14. 计算 $S_n = 1 + 11 + 111 + 1111 + \cdots + 1111\cdots111$(最后一项是 n 个 1)。提示：第 1 项 $T_1 = 1$；第 2 项 $T_2 = T_1 * 10 + 1$；…；第 n 项 $= T_{(n-1)} * 10 + 1$。n 是一个随机产生的 1～10(包括 1 和 10)中的正整数。其运行效果如图 4-15 所示。

```
第 8 天桃子数为： 1
第 7 天桃子数为： 4
第 6 天桃子数为： 10
第 5 天桃子数为： 22
第 4 天桃子数为： 46
第 3 天桃子数为： 94
第 2 天桃子数为： 190
第 1 天桃子数为： 382
```

图 4-14　猴子吃桃运行效果

```
n= 8 sn= 12345678
```

图 4-15　计算 S_n(其中 n 为随机数)的运行效果

4.13　案例研究：科学计算和数据分析

科学计算(Scientific Computing)泛指使用计算机科学基于数学建模和数值分析技术解决科学工程领域中问题的过程。科学计算是计算机科学、数学和工程的交叉学科。

随着 Python 语言生态环境的完善，众多科学计算和数据分析库(例如 NumPy、SciPy、Pandas、Matplotlib、IPython 等)出现，使得 Python 成为科学计算和数据分析的首选语言。

本章案例研究通过几个简单的应用例子引导读者进入科学计算的大门。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 5 章

序列数据类型



视频讲解

序列数据类型(bytes、bytearray、list、str 和 tuple)是 Python 内置的组合数据类型,可以实现复杂数据的处理。

5.1 Python 序列数据概述

5.1.1 数组

数组是一种数据结构,用于存储和处理大量的数据。将所有的数据存储在一个或多个数组中,然后通过索引下标访问并处理数组的元素,可实现复杂数据处理任务。

Python 语言没有提供直接创建数组的功能,但可以使用其内置的序列数据类型(例如列表)实现数组的功能。

5.1.2 Python 内置的序列数据类型

序列(sequence)数据类型是 Python 的基础数据结构,是一组有顺序的元素的集合。序列数据可以包含一个或多个元素(对象,元素也可以是其他序列数据),也可以是一个没有任何元素的空序列。

Python 内置的序列数据类型包括元组(tuple)、列表(list)、字符串(str)和字节数据(bytes 和 bytearray)。

元组也称为定值表,用于存储值固定不变的表。例如:

```
>>> s1 = (1,2,3)
>>> s1                    #输出:(1, 2, 3)
>>> s1[2]                 #输出:3
```

列表也称为表,用于存储其值可变的表。例如:

```
>>> s2 = [1,2,3]
>>> s2[2] = 4
>>> s2                    #输出:[1, 2, 4]
```

字符串是包括若干字符的序列数据,支持序列数据的基本操作。例如:

```
>>> s3 = "abc"
>>> s3 = "Hello, world!"
>>> s3[:5]                #字符串前 5 个字符.输出:'Hello'
```

字节序列数据是包括若干字节的序列。Python 抓取网页时返回的页面通常为 utf-8 编码的字节序列。字节序列和字符串可以直接相互转换。例如:

```
>>> s1 = b"abc"
```

```
>>> s1.decode("utf-8")        # 输出:'abc'
>>> s2 = "百度"
>>> s2.encode("utf-8")        # 输出:b'\xe7\x99\xbe\xe5\xba\xa6'
```

5.2　序列数据的基本操作

5.2.1　序列的长度、最大值、最小值、求和

通过内置函数 len()、max()、min()可以获取序列的长度、序列中元素的最大值、序列中元素的最小值。通过内置函数 sum()可以获取列表或元组中的各元素之和；如果有非数值元素，则导致 TypeError；对于字符串(str)和字节数据(bytes)，也将导致 TypeError。

【例 5.1】　序列数据的求和示例。

```
>>> t1 = (1,2,3,4)
>>> sum(t1)                   # 输出:10
>>> t2 = (1,'a',2)
>>> sum(t2)                   # TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> s = '1234'
>>> sum(s)                    # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

【例 5.2】　序列的长度、最大值、最小值操作示例。

>>> s = 'abcdefg'
>>> len(s)
7
>>> max(s)
'g'
>>> min(s)
'a'
>>> s2 = ''
>>> len(s2)
0
	>>> t = (10,2,3)
	>>> len(t)
	3
	>>> max(t)
	10
	>>> min(t)
	2
	>>> t2 = ()
	>>> len(t2)
	0
		>>> lst = [1,2,9,5,4]
		>>> len(lst)
		5
		>>> max(lst)
		9
		>>> min(lst)
		1
		>>> lst2 = []
		>>> len(lst2)
		0
			>>> b = b'ABCD'
			>>> len(b)
			4
			>>> max(b)
			68
			>>> min(b)
			65
			>>> b2 = b''
			>>> len(b2)
			0

5.2.2　序列的索引访问操作

序列表示可以通过索引下标访问的可迭代对象。用户可以通过整数下标访问序列 s 的元素。

```
s[i]                          # 访问序列 s 在索引 i 处的元素
```

索引下标从 0 开始，第 1 个元素为 s[0]，第 2 个元素为 s[1]，依此类推，最后一个元素为 s[len(s) − 1]。

如果索引下标越界，则导致 IndexError；如果索引下标不是整数，则导致 TypeError。例如：

```
>>> s = 'abc'
>>> s[0]                      # 输出:'a'
>>> s[3]                      # IndexError: string index out of range
>>> s['a']                    # TypeError: string indices must be integers
```

序列 s 的索引下标示意图如图 5-1 所示。

s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
'bonus'	-228	'purple'	'100'	19.84
s[0]	s[1]	s[2]	s[3]	s[4]

图 5-1　序列 s 的索引下标示意图

【例 5.3】　序列的索引访问示例。

```
>>> s = 'abcdef'
>>> s[0]
'a'
>>> s[2]
'c'
>>> s[-1]
'f'
>>> s[-3]
'd'
```

```
>>> t = ('a','e','i','o','u')
>>> t[0]
'a'
>>> t[1]
'e'
>>> t[-1]
'u'
>>> t[-5]
'a'
```

```
>>> lst = [1,2,3,4,5]
>>> lst[0]
1
>>> lst
[1, 2, 3, 4, 5]
>>> lst[2] = 'a'
>>> lst[-2] = 'b'
>>> lst
[1, 2, 'a', 'b', 5]
```

```
>>> b = b'ABCDEF'
>>> b[0]
65
>>> b[1]
66
>>> b[-1]
70
>>> b[-2]
69
```

5.2.3　序列的切片操作

通过切片(slice)操作可以截取序列 s 的一部分。切片操作的基本形式如下。

s[i:j]

或者

s[i:j:k]

其中,i 为序列开始下标(包含 s[i]);j 为序列结束下标(不包含 s[j]);k 为步长。如果省略 i,则从下标 0 开始;如果省略 j,则直到序列结束为止;如果省略 k,则步长为 1。

注意: 下标也可以为负数。如果截取范围内没有数据,则返回空元组;如果超过下标范围,则不报错。

【例 5.4】　序列的切片操作示例。

```
>>> s = 'abcdef'
>>> s[1:3]
'bc'
>>> s[3:10]
'def'
>>> s[8:2]
''
>>> s[:]
'abcdef'
>>> s[:2]
'ab'
>>> s[::2]
'ace'
>>> s[::-1]
'fedcba'
```

```
>>> t = ('a','e','i','o','u')
>>> t[-2:-1]
('o',)
>>> t[-2:]
('o', 'u')
>>> t[-99:-5]
()
>>> t[-99:-3]
('a', 'e')
>>> t[::]
('a', 'e', 'i', 'o', 'u')
>>> t[1:-1]
('e', 'i', 'o')
>>> t[1::2]
('e', 'o')
```

```
>>> lst = [1,2,3,4,5]
>>> lst[:2]
[1, 2]
>>> lst[:1] = []
>>> lst
[2, 3, 4, 5]
>>> lst[:2]
[2, 3]
>>> lst[:2] = 'a'
>>> lst[1:] = 'b'
>>> lst
['a', 'b']
>>> del lst[:1]
>>> lst
['b']
```

```
>>> b = b'ABCDEF'
>>> b[2:2]
b''
>>> b[0:1]
b'A'
>>> b[1:2]
b'B'
>>> b[2:2]
b''
>>> b[-1:]
b'F'
>>> b[-2:-1]
b'E'
>>> b[0:len(b)]
b'ABCDEF'
```

5.2.4 序列的连接和重复操作

通过连接操作符+可以连接两个序列(s1 和 s2),形成一个新的序列对象;通过重复操作符 * 可以重复一个序列 n 次(n 为正整数)。序列连接和重复操作的基本形式如下。

s1 + s2 或者 s * n 或者 n * s

连接操作符+和重复操作符 * 也支持复合赋值运算,即+=和 *=。

【例 5.5】 序列的连接和重复操作示例。

```
>>> s1 = 'abc'
>>> s2 = 'xyz'
>>> s1 + s2
'abcxyz'
>>> s1 * 3
'abcabcabc'
>>> s1  +=  s2
>>> s1
'abcxyz'
>>> s2  * =  2
>>> s2
'xyzxyz'
```

```
>>> t1 = (1,2)
>>> t2 = ('a','b')
>>> t1 + t2
(1, 2, 'a', 'b')
>>> t1 * 2
(1, 2, 1, 2)
>>> t1  +=  t2
>>> t1
(1, 2, 'a', 'b')
>>> t2  * =  2
>>> t2
('a', 'b', 'a', 'b')
```

```
>>> lst1 = [1,2]
>>> lst2 = ['a','b']
>>> lst1 +  lst2
[1, 2, 'a', 'b']
>>> 2  *  lst2
['a', 'b', 'a', 'b']
>>> lst1  +=  lst2
>>> lst1
[1, 2, 'a', 'b']
>>> lst2  * = 2
>>> lst2
['a', 'b', 'a', 'b']
```

```
>>> b1 = b'ABC'
>>> b2 = b'XYZ'
>>> b1 + b2
b'ABCXYZ'
>>> b1 * 3
b'ABCABCABC'
>>> b1 += b2
>>> b1
b'ABCXYZ'
>>> b2 *  = 2
>>> b2
b'XYZXYZ'
```

5.2.5 序列的成员关系操作

用户可以通过下列方式之一判断元素 x 是否存在于序列 s 中:

- **x in s** #如果为 True,则表示存在
- **x not in s** #如果为 True,则表示不存在
- **s.count(x)** #返回 x 在 s(指定范围[start,end))中出现的次数
- **s.index(x[, i[, j]])** #返回 x 在 s(指定范围[i, j))中第一次出现的下标

其中,指定范围[i, j)表示从下标 i(包括,默认为 0)开始,到下标 j 结束(不包括,默认为 len(s))。

对于 s.index(value, [start, [stop]])方法,如果找不到,则导致 ValueError。例如:

```
>>> 'To be or not to be, this is a question'.index('123')    #ValueError: substring not found
```

【例 5.6】 序列中元素存在性的判断示例。

```
>>> s = 'Good, better,
best!'
>>> 'o' in s
True
>>> 'g' not in s
True
>>> s.count('e')
3
>>> s.index('e', 10)
10
```

```
>>> t = ('r', 'g', 'b')
>>> 'r' in t
True
>>> 'y' not in t
True
>>> t.count('r')
1
>>> t.index('g')
1
```

```
>>> lst = [1,2,3,2,1]
>>> 1 in lst
True
>>> 2 not in lst
False
>>> lst.count(1)
2
>>> lst.index(3)
2
```

```
>>> b = b'Oh, Jesus!'
>>> b'O' in b
True
>>> b'o' not in b
True
>>> b.count(b's')
2
>>> b.index(b's')
6
```

5.2.6 序列的比较运算操作

两个序列支持比较运算符（<、<=、==、!=、>=、>），字符串比较运算按顺序逐个元素进行比较。

【例 5.7】 序列的比较运算示例。

```
>>> s1 = 'abc'
>>> s2 = 'abc'
>>> s3 = 'abcd'
>>> s4 = 'cba'
>>> s1 > s4
False
>>> s2 <=  s3
True
>>> s1  ==  s2
True
>>> s1  !=  s3
True
>>> 'a'> 'A'
True
>>> 'a'>=  ''
True
```

```
>>> t1 = (1,2)
>>> t2 = (1,2)
>>> t3 = (1,2,3)
>>> t4 = (2,1)
>>> t1 < t4
True
>>> t1 <=  t2
True
>>> t1  ==  t3
False
>>> t1  !=  t2
False
>>> t1 >=  t3
False
>>> t4 > t3
True
```

```
>>> s1 = ['a', 'b']
>>> s2 = ['a', 'b']
>>> s3 = ['a', 'b', 'c']
>>> s4 = ['c', 'b', 'a']
>>> s1 < s2
False
>>> s1 <= s2
True
>>> s1 == s2
True
>>> s1!= s3
True
>>> s1 >= s3
False
>>> s4 > s3
True
```

```
>>> b1 = b'abc'
>>> b2 = b'abc'
>>> b3 = b'abcd'
>>> b4 = b'ABCD'
>>> b1 < b2
False
>>> b1 <= b2
True
>>> b1 == b2
True
>>> b1 >= b3
False
>>> b3!= b4
True
>>> b4 > b3
False
```

5.2.7 序列的排序操作

通过内置函数 sorted()可以返回序列的排序列表。通过类 reversed 构造函数可以返回序列的反序迭代器。内置函数 sorted()的形式如下。

```
sorted(iterable, key = None, reverse = False)     #返回序列的排序列表
```

其中，key 是用于计算比较键值的函数（带一个参数），例如 key=str. lower。如果 reverse=True，则反向排序。

【例 5.8】 序列的排序操作示例。

```
>>> s1 = 'axd'
>>> sorted(s1)
['a', 'd', 'x']
>>> s2 = (1,4,2)
```

```
>>> sorted(s2)
[1, 2, 4]
>>> sorted(s2,reverse = True)
[4, 2, 1]
```

```
>>> s3 = 'abAC'
>>> sorted(s3, key = str.lower)
['a', 'A', 'b', 'C']
```

5.2.8 内置函数 all()和 any()

通过内置函数 all()和 any()可以判断序列的元素是否全部或部分为 True，函数形式如下。

- **all(iterable)** #如果序列的所有值都为 True，返回 True；否则返回 False
- **any(iterable)** #如果序列的任意值为 True，返回 True；否则返回 False

例如：

```
>>> any((1, 2, 0))
True
```

```
>>> all([1, 2, 0])
False
```

5.2.9 序列的拆分

1. 变量个数和序列长度相等

使用赋值语句可以将序列值拆分，然后赋值给多个变量，形式如下。

变量 1, 变量 2,…, 变量 n = 序列或可迭代对象

若变量个数和序列的元素个数不一致，将导致 ValueError。例如：

```
>>> a, b = (1, 2)
>>> a, b                        #输出:(1, 2)
>>> a, b, c = (1, 2)            #ValueError: not enough values to unpack (expected 3, got 2)
>>> data = (1001, '张三', (80, 79, 92))
>>> sid, name, scores = data
>>> scores                      #输出:(80, 79, 92)
>>> sid, name, (chinese, math, english) = data
>>> math                        #输出:79
```

2. 变量个数和序列长度不等

如果序列长度未知，可以使用 * 元组变量，将多个值作为元组赋值给元组变量。在一个赋值语句中，* 元组变量只允许出现一次，否则将导致 SyntaxError。例如：

```
>>> first, *middles, last = range(10)
>>> middles                     #输出:[1, 2, 3, 4, 5, 6, 7, 8]
>>> first, second, third, *lasts = range(10)
>>> lasts                       #输出:[3, 4, 5, 6, 7, 8, 9]
>>> *firsts, last3, last2, last1 = range(10)
>>> firsts                      #输出:[0, 1, 2, 3, 4, 5, 6]
>>> first, *middles, last = sorted([70, 85, 89, 88, 86, 95, 89])  #去掉最高分和最低分
>>> sum(middles)/len(middles)   #计算去掉最高分和最低分后的平均值.输出:87.4
```

3. 使用临时变量_

如果只需要部分数据，序列的其他位置可以使用临时变量_。例如：

```
>>> _, b, _ = (1, 2, 3)
>>> b                           #输出:2
>>> record = ('Zhangsan', 'szhang@abc.com', '021-62232333', '13912349876')
>>> name, _, *phones = record
>>> phones                      #输出:['021-62232333', '13912349876']
```

5.3 元　　组

元组(tuple)是一组有序序列，包含零个或多个对象引用。元组和列表十分类似，但元组是不可变的对象，即用户不能修改、添加或删除元组中的项目(可以访问元组中的项目)。

5.3.1 使用元组字面量创建元组实例对象

使用元组字面量可以创建元组实例对象。元组字面量采用在圆括号中以逗号分隔的项目定义，圆括号可以省略。其基本形式如下。

x1, [x2,…, xn] 或者 (x1, [x2,…, xn])

其中，x1，x2，…，xn 为任意对象。

【例 5.9】 使用元组字面量创建元组实例对象的示例。

```
>>> t1 = 1,2,3; t2 = (); t3 = 1,; i1 = (1); t4 = 'a','b','c'; t5 = 2.0,
>>> print(t1,t2,t3,t4,t5,i1)     #输出:(1, 2, 3) () (1,) ('a', 'b', 'c') (2.0,) 1
```

注意：如果元组中只有一个项目，后面的逗号不能省略。这是因为 Python 解释器把(x1)解释为 x1，例如将(1)解释为整数 1，将(1,)解释为元组。

5.3.2 使用 tuple 对象创建元组实例对象

用户也可以通过创建 tuple 对象来创建元组，其基本形式如下。

- **tuple()**   #创建一个空元组
- **tuple(iterable)**   #创建一个元组，包含的项目为可枚举对象 iterable 中的元素

【例 5.10】 使用 tuple 对象创建元组实例对象的示例。

```
>>> t1 = tuple(); t2 = tuple("abc")
>>> t3 = tuple([1,2,3]); t4 = tuple(range(3))
>>> print(t1,t2,t3,t4)           #输出:() ('a', 'b', 'c') (1, 2, 3) (0, 1, 2)
```

5.3.3 元组的序列操作

元组支持序列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求元组的长度、最大值、最小值等。

【例 5.11】 元组的序列操作示例。

```
>>> t1 = (1,2,3,4,5,6,7,8,9,10)
>>> len(t1)                      #输出:10
>>> max(t1)                      #输出:10
>>> sum(t1)                      #输出:55
```

5.4 列　　表

列表(list)是一组有序项目的数据结构。在创建一个列表后，用户可以访问、修改、添加或删除列表中的项目，即列表是可变的数据类型。在 Python 中没有数组，可以使用列表代替。

5.4.1 使用列表字面量创建列表实例对象

使用列表字面量可以创建列表实例对象。列表字面量采用在方括号中以逗号分隔的项目定义，其基本形式如下。

[x1[,x2, …, xn]]

【例 5.12】 使用列表字面量创建列表实例对象的示例。

```
>>> l1 = []; l2 = [1]; l3 = ["a","b","c"]
>>> print(l1,l2,l3)              #输出:[] [1] ['a', 'b', 'c']
```

5.4.2 使用 list 对象创建列表实例对象

用户也可以通过创建 list 对象来创建列表，其基本形式如下。

- **list()**   #创建一个空列表
- **list(iterable)**   #创建一个列表，包含的项目为可枚举对象 iterable 中的元素

【例 5.13】　使用 list 对象创建列表实例对象的示例。

```
>>> l1 = list(); l2 = list("abc"); l3 = list(range(3))
>>> print(l1,l2,l3)              #输出:[] ['a', 'b', 'c'] [0, 1, 2]
```

5.4.3　列表的序列操作

列表支持序列的基本操作,包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作,以及求列表的长度、最大值、最小值等。

列表是可变对象,故用户可以改变列表对象中元素的值,也可以通过 del 删除某元素。

```
s[下标] = x                      #设置列表元素,x 为任意对象
del s[下标]                      #删除列表元素
```

列表是可变对象,故用户可以改变其切片的值,也可以通过 del 删除切片。

```
s[i:j] =x                        #设置列表内容,x 为任意对象,也可以是元组、列表
del s[i:j]                       #移去列表的一系列元素,等同于 s[i:j] =[]
s[i:j] =[]                       #移去列表的一系列元素
```

【例 5.14】　列表的序列操作示例。

```
>>> s = [1,2,3,4,5,6]
>>> s[1] = 'a'
>>> s
[1, 'a', 3, 4, 5, 6]
>>> s[2] = []
>>> s
[1, 'a', [], 4, 5, 6]
>>> del s[3]
>>> s
[1, 'a', [], 5, 6]
>>> s[:2]
[1, 'a']
>>> s[2:3] = []
>>> s
[1, 'a', 5, 6]
>>> s[:1] = []
>>> s
['a', 5, 6]
>>> s[:2] = 'b'
>>> s
['b', 6]
>>> del s[:1]
>>> s
[6]
```

5.4.4　list 对象的方法

列表是可变对象,其包含的主要方法如表 5-1 所示。假设该表中的示例基于 s=[1,3,2]。

表 5-1　列表对象的主要方法

方　法	说　明	示　例
s.append(x)	把对象 x 追加到列表 s 的尾部	s.append('a')　#s=[1, 3, 2, 'a'] s.append([1,2])　#s=[1, 3, 2, 'a', [1, 2]]
s.clear()	删除所有元素,相当于 del s[:]	s.clear()　#s=[]
s.copy()	复制列表	s1=s.copy()　#s1= s=[1,3,2] id(s),id(s1)　#(3143376592008, 3143376591496)
s.extend(t)	把序列 t 附加到 s 的尾部	s.extend([4])　#s=[1, 3, 2, 4] s.extend('ab')　#s=[1, 3, 2, 4, 'a', 'b']
s.insert(i,x)	在下标 i 位置插入对象 x	s.insert(1,4)　#s=[1, 4, 3, 2] s.insert(8,5)　#s=[1, 4, 3, 2, 5]
s.pop([i])	返回并移除下标 i 位置的对象,当省略 i 时为最后对象。若超出下标,将导致 IndexError	s.pop()　#输出 2。s=[1, 3] s.pop(0)　#输出 1。s=[3]
s.remove(x)	移除列表中第一个出现的 x。若对象不存在,将导致 ValueError	s.remove(1)　#s=[3, 2] s.remove('a')　#ValueError: list.remove(x): x not in list

续表

方　法	说　明	示　例
s.reverse()	列表反转	s.reverse()　　#s=[2,3,1]
s.sort()	列表排序	s.sort()　　#s=[1,2,3]

5.4.5 列表解析表达式

使用列表解析表达式可以简单、高效地处理一个可迭代对象,并生成结果列表。列表解析表达式的形式如下。

- [expr for i_1 in 序列1… for i_N in 序列N]　　#迭代序列中的所有内容,并计算生成列表
- [expr for i_1 in 序列1… for i_N in 序列N if cond_expr]#按条件迭代,并计算生成列表

表达式expr使用每次迭代的内容i_1～i_N计算生成一个列表。如果指定了条件表达式cond_expr,则只有满足条件的元素参与迭代。

【例5.15】 列表解析表达式示例。

```
>>> [i**2 for i in range(10)]                                  #平方值
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [(i,i**2) for i in range(10)]                              #(序号, 平方值)
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
>>> [i for i in range(10) if i%2==0]                           #取偶数
[0, 2, 4, 6, 8]
>>> [(x, y, x*y) for x in range(1, 4) for y in range(1, 4) if x>=y]   #二重循环
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

5.5 字　符　串

字符串(str)为有序的字符集合,即字符序列。str对象是不可变对象。

5.5.1 字符串的序列操作

字符串支持序列的基本操作,包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作,以及求字符串的长度、最大值、最小值等。

通过len(s)可以获取字符串s的长度,如果其长度为0,则为空字符串。

【例5.16】 字符串的序列操作示例。

```
>>> s1 = 'abcxyz'
>>> len(s1)
6
>>> s1[3:]
'xyz'
>>> s2 = '123'
>>> s1 > s2
True
>>> 3 * s2
'123123123'
>>> max(s1)
'z'
```

5.5.2 字符串编码

在默认情况下,Python字符串采用utf-8编码。在创建字符串时,用户也可以指定其编码方式:

str(object = b'', encoding = 'utf-8', errors = 'strict')　#按指定编码根据字节码对象创建str对象

其中,object为字节码对象(bytes或bytearray);encoding为编码;errors为错误控制。该构

造函数的结果等同于 bytes 对象 b 的对象方法：

```
b.decode(encoding, errors)          #把字节码对象 b 解码为对应编码的字符串
```

与之相对应，用户也可以把字符串对象 s 编码为字节对象：

```
s.encode(encoding = "utf - 8", errors = "strict")       #把字符串对象 s 编码为字节对象
```

【例 5.17】 字符串编码和解码示例。

```
>>> s1 = 'Sample!例子!'
>>> b1 = s1.encode(encoding = 'cp936')
>>> b1
b'Sample!\xc0\xfd\xd7\xd3\xa3\xa1'
>>> b1.decode(encoding = 'cp936')
'Sample!例子!'
```

```
>>> b1.decode()      #默认编码为 utf-8,出错
Traceback(most recent call last):
  File "<pyshell#56>", line 1, in <module>
    b1.decode()
UnicodeDecodeError: 'utf-8' codec can't decode
byte 0xc0 in position 7: invalid start byte
```

5.5.3　字符串的格式化

1. %运算符形式

Python 支持类似于 C 语言的 printf 格式化输出，采用如下形式。

```
格式字符串 % (值 1, 值 2, …)                    #兼容 Python 2 的格式,不建议使用
```

格式字符串与 C 语言中的 printf 格式字符串基本相同。格式字符串由固定文本和格式说明符混合组成。格式说明符的语法如下。

```
%[(key)][flags][width][.precision][Length]type
```

其中：key(可选)为映射键(适用于映射的格式化，例如'%(lang)s')；flags(可选)为修改输出格式的字符集；width(可选)为最小宽度，如果为 *，则使用下一个参数值；precision(可选)为精度，如果为 *，则使用下一个参数值；Length 为修饰符(h、l 或 L，可选)，Python 忽略该字符；type 为格式化类型字符。例如：

```
>>> '结果:%f' % 88                                    #输出:'结果:88.000000'
>>> '姓名:%s, 年龄:%d, 体重:%3.2f' % ('张三', 20, 53)
'姓名:张三, 年龄:20, 体重:53.00'
>>> '%(lang)s has %(num)03d quote types.' % {'lang':'Python', 'num': 2}
'Python has 002 quote types.'
>>> '%0*.*f' % (10, 5, 88)                            #输出:'0088.00000'
```

格式字符串的标识符(flags)如下。

- '0'：数值类型格式化结果左边用 0 填充。
- '-'：结果左对齐。
- ' '：对于正值，结果中将包括一个前导空格。
- '+'：数值结果总是包括一个符号('+'或'-')。
- '#'：使用另一种转换方式。

格式化类型字符(type)如下。

- %d 或%i：有符号整数(十进制)。
- %o：有符号整数(八进制)。
- %u：同%d，已过时。
- %x：有符号整数(十六进制，小写字符)，当标识符为'#'时输出前缀'0x'。

- %X：有符号整数（十六进制，大写字符），当标识符为'#'时输出前缀'0X'。
- %e：浮点数字（科学记数法，小写 e），当标识符为'#'时总是带小数点。
- %E：浮点数字（科学记数法，大写 E），当标识符为'#'时总是带小数点。
- %f 或%F：浮点数字（用小数点符号），当标识符为'#'时总是带小数点。
- %g：浮点数字（根据值的大小采用%e 或%f），当标识符为'#'时总是带小数点，保留后面的 0。
- %G：浮点数字（根据值的大小采用%E 或%F），当标识符为'#'时总是带小数点，保留后面的 0。
- %c：字符及其 ASCII 码。
- %r：字符串，使用转换函数 repr()，当标识符为'#'且指定 precision 时截取 precision 个字符。
- %s：字符串，使用转换函数 str()，当标识符为'#'且指定 precision 时截取 precision 个字符。
- %a：字符串，使用转换函数 ascii()，当标识符为'#'且指定 precision 时截取 precision 个字符。

2. format 内置函数

format 内置函数的基本形式如下。

```
• format(value)                    #等同于 str(value)
• format(value, format_spec)       #等同于 type(value).__format__(format_spec)
```

格式说明符（format_spec）的基本格式如下。

```
[[fill]align][sign][#][0][width][,][.precision][type]
```

其中，fill（可选）为填充字符，可以为除{}以外的任何字符；align 为对齐方式，包括"<"（左对齐）、">"（右对齐）、"="（填充位于符号和数字之间，例如'+000000120'）、"^"（居中对齐）；sign（可选）为符号字符，包括"+"（正数）、"-"（负数）、" "（正数带空格，负数带-）；'#'（可选）为使用另一种转换方式；'0'（可选）为数值类型格式化结果左边用 0 填充；width（可选）是最小宽度；precision（可选）是精度；type 是格式化类型字符。

格式化类型字符（type）如下。

- b：二进制数。
- c：字符，整数转换为对应的 unicode。
- d：十进制数。
- o：八进制数。
- x：十六进制数，小写字符，当标识符为'#'时输出前缀'0x'。
- X：十六进制数，大写字符，当标识符为'#'时输出前缀'0X'。
- e：浮点数字（科学记数法，小写 e），当标识符为'#'时总是带小数点。
- E：浮点数字（科学记数法，大写 E），当标识符为'#'时总是带小数点。
- f 或 F：浮点数字（用小数点符号），当标识符为'#'时总是带小数点。
- g：浮点数字（根据值的大小采用 e 或 f），当标识符为'#'时总是带小数点，保留后面的 0。
- G：浮点数字（根据值的大小采用 E 或 F），当标识符为'#'时总是带小数点，保留后面的 0。

- n：数值，使用本地千位分隔符。
- s：字符串，使用转换函数 str()，当标识符为 '#' 且指定 precision 时截取 precision 个字符。
- %：百分比。
- _：十进制千分位分隔符或者二进制 4 位分隔符（Python 3.6 的新增功能）。

例如：

```
>>> format(81.2, "0.5f")                #输出：'81.20000'
>>> format(81.2, "%")                   #输出：'8120.000000%'
>>> format(1000000, "_")                #输出：1_000_000'
>>> format(1024, "_b")                  #输出：'100_0000_0000'
```

3. 字符串的 format 方法

字符串的 format 方法的基本形式如下。

- **str.format(格式字符串, 值 1, 值 2, …)**　　#类方法
- **格式字符串.format(值 1, 值 2, …)**　　#对象方法
- **格式字符串.format_map(mapping)**

格式字符串由固定文本和格式说明符混合组成。格式说明符的语法如下。

```
{[索引和键]:format_spec}
```

其中，可选的索引对应于要格式化参数值的位置，可选的键对应于要格式化的映射的键；格式化说明符(format_spec)同 format 内置函数。例如：

```
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(100)
'int: 100; hex: 64; oct: 144; bin: 1100100'
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(100)
'int: 100; hex: 0x64; oct: 0o144; bin: 0b1100100'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')      #输出：'c, b, a'
>>> str.format_map('{name:s},{age:d},{weight:3.2f}', {'name':'Mary', 'age':20, 'weight':49})
'Mary,20,49.00'
```

5.6　字 节 序 列

字节序列（bytes 和 bytearray）是由 8 位字节数据组成的序列数据类型，即 $0 \leqslant x < 256$ 的整数序列。Python 内置的字节序列数据类型包括 bytes（不可变对象）、bytearray（可变对象）和 memoryview。

5.6.1　bytes 常量

使用字母 b 加单引号或双引号括起来的内容是 bytes 常量。Python 解释器自动创建 bytes 型对象实例。bytes 常量与字符串的定义方式类似。

(1) 单引号(b'　')：包含在单引号中的字符串，其中可以包含双引号。

(2) 双引号(b"　")：包含在双引号中的字符串，其中可以包含单引号。

(3) 三单引号(b'''　''')：包含在三单引号中的字符串，可以跨行。

(4) 三双引号(b"""　""")：包含在三双引号中的字符串，可以跨行。

注意：在引号中只能包含 ASCII 码字符，否则将导致 SyntaxError。例如：

```
>>> b'张'              #SyntaxError: bytes can only contain ASCII literal characters
```

【例 5.18】 bytes 常量示例。

```
>>> b'abc'
b'abc'
>>> b'abc\'x\''
b"abc'x'"
>>> b'abc"x"'
b'abc"x"'
>>> x = b'c:\\Python33'
>>> x
b'c:\\Python33'
```

```
>>> b"xyz"
b'xyz'
>>> b"x\tyz"
b'x\tyz'
>>> print(b"x\tyz")
b'x\tyz'
>>> print(b"x'y'z")
b"x'y'z"
>>> print(b"x\ny")
b'x\ny'
```

```
>>> s1 = '''a
\tb
\tc
\td'''
>>> s1 = b'''a
\tb
\tc
\td'''
>>> s1
b'a\n\tb\n\tc\n\td'
```

```
>>> s2 = b"""
She said:
"Yes!"
"""
>>> s2
b'\nShe said:\n"Yes!"\n'
>>> print(s2)
b'\nShe said:\n"Yes!"\n'
```

5.6.2 创建 bytes 对象

创建 bytes 类型的对象实例的基本形式如下。

- **bytes()** # 创建空 bytes 对象
- **bytes(n)** # 创建长度为 n(整数)的 bytes 对象,各字节为 0
- **bytes(iterable)** # 创建 bytes 对象,使用 iterable 中的字节整数
- **bytes (object)** # 创建 bytes 对象,复制 object 字节数据
- **bytes([source[, encoding[, errors]]])** # 创建 bytes 对象

如果 iterable 中包含非 $0 \leqslant x < 256$ 的整数,将导致 ValueError。

【例 5.19】 创建 bytes 对象示例。

```
>>> bytes()
b''
>>> bytes(2)
b'\x00\x00'
```

```
>>> bytes((1,2,3))
b'\x01\x02\x03'
>>> bytes('abc', 'utf-8')
b'abc'
```

```
>>> bytes((123, 456))
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in <module>
    bytes((123, 456))
ValueError: bytes must be in range(0, 256)
```

5.6.3 创建 bytearray 对象

创建 bytearray 类型的对象实例的基本形式如下。

- **bytearray()** # 创建空 bytearray 对象
- **bytearray(n)** # 创建长度为 n(整数)的 bytearray 对象,各字节为 0
- **bytearray(iterable)** # 创建 bytearray 对象,使用 iterable 中的字节整数
- **bytearray(object)** # 创建 bytearray 对象,复制 object 字节数据
- **bytearray([source[, encoding[, errors]]])** # 创建 bytearray 对象

如果 iterable 中包含非 $0 \leqslant x < 256$ 的整数,将导致 ValueError。

【例 5.20】 创建 bytearray 对象示例。

```
>>> bytearray()
bytearray(b'')
>>> bytearray(2)
bytearray(b'\x00\x00')
```

```
>>> bytearray((1,2,3))
bytearray(b'\x01\x02\x03')
>>> bytearray('abc', 'utf-8')
bytearray(b'abc')
```

```
>>> bytearray((123,456))
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in
<module>
    bytearray((123,456))
ValueError: byte must be in range(0,
256)
```

5.6.4 bytes 和 bytearray 的序列操作

bytes 和 bytearray 支持序列的基本操作，包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作，以及求序列的长度、最大值、最小值等。

bytes 和 bytearray 一般基于 ASCII 字符串，故 bytes 和 bytearray 基本上支持 str 对象的类似方法，但不支持 str.encode()（把字符串转换为 bytes 对象）、str.format()/str.format_map()（字符串格式化）、str.isidentifier()/str.isnumeric()/str.isdecimal()/str.isprintable()（这些判断无意义）。

注意： bytes 和 bytearray 的方法不接受字符串参数，只接受 bytes 和 bytearray 参数，否则将导致 TypeError。

【例 5.21】 字节的序列操作示例。

```
>>> b1 = b"abc"
>>> b1.replace(b'a',b'f')          #输出:b'fbc'
>>> b1.replace('b','g')            # TypeError: a bytes-like object is required, not 'str'
```

5.6.5 字节编码和解码

字符串可以通过 str.encode()方法编码为字节码，通过 bytes 和 bytearray 的 decode()方法解码为字符串。

【例 5.22】 字节编码和解码示例。

```
>>> s = '好好学习'
>>> b = s.encode()
>>> b                              #输出:b'\xe5\xa5\xbd\xe5\xa5\xbd\xe5\xad\xa6\xe4\xb9\xa0'
>>> b.decode()                     #输出:'好好学习'
```

5.7 复　习　题

一、选择题

1. Python 语句 print(type([1, 2, 3, 4]))的运行结果是________。

   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>

2. Python 语句 print(type((1, 2, 3, 4)))的运行结果是________。

   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>

3. Python 语句 print(type({1, 2, 3, 4}))的运行结果是________。

   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>

4. Python 语句序列“a = (1,2,3,None,(),[],); print(len(a))”的运行结果是________。

   A. 4　B. 5　C. 6　D. 7

5. Python 语句序列“nums = set([1,2,2,3,3,3,4]); print(len(nums))”的运行结果是________。

   A. 1　B. 2　C. 4　D. 7

6. Python 语句序列“s= 'hello'; print(s[1:3])”的运行结果是________。

A. hel　　B. he　　C. ell　　D. el

7. Python 语句序列“s1 = [4, 5, 6]; s2 = s1; s1[1] = 0; print(s2)”的运行结果是________。

A. [4,5,6]　　B. [0,5,6]　　C. [4,0,6]　　D. 以上都不对

8. Python 语句序列“d={1: 'a', 2: 'b', 3: 'c'}; print(len(d))”的运行结果是________。

A. 0　　B. 1　　C. 3　　D. 6

9. Python 语句序列“a = [1, 2, 3, None, (), [],]; print(len(a))”的运行结果是________。

A. 语法错　　B. 4　　C. 5　　D. 6

10. Python 语句 print('\x48\x41!')的运行结果是________。

A. '\x48\x41!'　　B. 4841!　　C. 4841　　D. HA!

11. Python 语句序列“s={'a',1,'b',2}; print(s['b'])”的运行结果是________。

A. 语法错　　B. 'b'　　C. 1　　D. 2

12. Python 语句 print(r"\nGood")的运行结果是________。

A. 新行和字符串 Good　　B. r"\nGood"

C. \nGood　　D. 字符 r、新行和字符串 Good

二、填空题

1. Python 语句序列“fruits=['apple','banana','pear']; print(fruits[-1][-1])”的运行结果是________。

2. Python 语句序列“fruits=['apple','banana','pear']; print(fruits. index('apple'))”的运行结果是________。

3. Python 语句序列“fruits=['apple','banana','pear']; print('Apple' in fruits)”的运行结果是________。

4. Python 语句 print(sum(range(10)))的运行结果是________。

5. Python 语句 print('%d%%%d' %(3/2, 3%2))的运行结果是________。

6. Python 语句序列“s = [1, 2, 3, 4]; s. append([5,6]); print(len(s))”的运行结果是________。

7. Python 语句序列“s1=[1,2,3,4]; s2=[5,6,7]; print(len(s1+s2))”的运行结果是________。

8. Python 语句序列“print(tuple(range(2)), list(range(2)))”的运行结果是________。

9. Python 语句序列“print(tuple([1,2,3]), list([1,2,3]))”的运行结果是________。

10. Python 列表解析表达式[i for i in range(5) if i%2!=0]和[i ** 2 for i in range(3)]的值分别为________。

11. Python 语句“first, * middles, last = range(6)”执行后，middles 的值为________；语句“first, second, third, * lasts = range(6)”执行后，lasts 的值为________；语句“* firsts, last3, last2, last1 = range(6)”执行后，firsts 的值为________；语句“first, * middles, last = sorted([86, 85, 99, 88, 60, 95, 96])”执行后，sum(middles)/len(middles)的值为________。

12. 在 Python 中设有 s=('a', 'b', 'c', 'd', 'e')，则 s[2]值为________；s[2:4]值为________；s[: 3]值为________；s[3:]值为________；s[1: :2]值为________；s[-2]值为

________；s[：：－1]值为________；s[－2：－1]值为________；s[－2：]值为________；s[－99：－5]值为________；s[－99：－3]值为________；s[：：]值为________；s[1：－1]值为________。

13. 在 Python 中设有 s＝[1，2，3，4，5，6]，则 max(s)值为________；min(s)值为________；语句序列“s[：1]＝[]；s[：2]＝'a'；s[2：]＝'b'；s[2:3]＝['x'，'y']；del s[：1]”执行后，s 值为________。

14. 在 Python 中设有 s＝['a'，'b']，则语句序列“s. append([1，2])；s. extend('34')；s. extend([5，6])；s. insert(1，7)；s. insert(10，8)；s. pop()；s. remove('b')；s[3：]＝[]；s. reverse()”执行后，s 值为________。

三、思考题

1. 在 Python 中如何实现 tuple 和 list 的转换？

2. 阅读下面的 Python 语句，请问输出结果是什么？

```
n = int(input("请输入图形的行数: "))
for i in range(n,0, - 1):
      print(" ".rjust(20 - i),end = '')
      for j in range(2 * i - 1):print(" * ",end = '')
      print("\n")
for i in range( 1, n):
      print(" ".rjust(19 - i),end = '')
      for j in range(2 * i + 1):print(" * ",end = '')
      print("\n")
```

3. 阅读下面的 Python 语句，请问输出结果是什么？

```
n = int(input("请输入上(或下)三角的行数: "))
for i in range(0,n):
      print(" ".rjust(19 - i),end = '')
      for j in range(2 * i + 1):print(" * ",end = '')
      print("\n")
for i in range(n - 1, 0, - 1):
      print(" ".rjust(20 - i),end = '')
      for j in range(2 * i - 1):print(" * ",end = '')
      print("\n")
```

4. 阅读下面的 Python 语句，请问输出结果是什么？

```
daysOfWeek = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday']
months = ['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
print("DAYS: %s, MONTHS %s" % (daysOfWeek, months))
```

5. 阅读下面的 Python 语句，请问输出结果是什么？

```
names1 = ['Amy', 'Bob', 'Charlie', 'Daling']
names2 = names1; names3 = names1[:]
names2[0] = 'Alice';names3[1] = 'Ben'
sum = 0
for ls in (names1, names2, names3):
    if ls[0] == 'Alice': sum += 1
    if ls[1] == 'Ben': sum += 2
print(sum)
```

5.8 上机实践

1. 完成本章中的例 5.1～例 5.22,熟悉 Python 语言序列数据类型(bytes、bytearray、list、str 和 tuple)的运算和操作。

2. 统计所输入字符串中单词的个数,单词之间用空格分隔。其运行效果如图 5-2 所示。

```
请输入字符串: The quick brown fox jumps over the lazy dog.
其中的单词总数有:  9
```

图 5-2 统计单词运行效果

3. 编写程序,删除一个 list 里面的重复元素。

提示:

可以利用 s.append(x)方法把对象 x 追加到列表 s 的尾部。

4. 编写程序,求列表 s=[9,7,8,3,2,1,55,6]中的元素个数、最大值、最小值,以及元素之和、平均值。请思考有哪几种实现方法?

提示:

可以分别利用 for 循环、while 循环、直接访问列表元素(for i in s…)、间接访问列表元素(for i in range(0,len(s))…)、正序访问(i=0; while i<len(s)…)、反序访问(i=len(s)-1; while i>=0…)以及 while True: …break 等方法。

5. 编写程序,将列表 s=[9,7,8,3,2,1,5,6]中的偶数变成它的平方,奇数保持不变。其运行效果如图 5-3 所示。

提示:

可以利用“if (s[i] % 2) ==0: ...”的语句形式判断列表中的第 i 个元素是否为偶数。

6. 编写程序,输入字符串,将其每个字符的 ASCII 码形成列表并输出,运行效果如图 5-4 所示。

```
变换前,s= [9, 7, 8, 3, 2, 1, 5, 6]
变换后,s= [9, 7, 64, 3, 4, 1, 5, 36]
```

图 5-3 奇数、偶数运行效果

```
请输入一个字符串:ABCDE123
[65, 66, 67, 68, 69, 49, 50, 51]
```

图 5-4 ASCII 码列表运行效果

提示:

(1) 使用 ord(s[i])方法将字符转换为对应的 Unicode 码。

(2) 使用 s.append(x)方法将对象 x 追加到列表 s 的尾部。

5.9 案例研究:猜单词游戏

本章案例研究通过一个简单的游戏案例帮助读者使用数据结构和算法实现基本的游戏人工智能,从而加深了解 Python 数据结构和基本算法流程。

“猜单词游戏”使用元组或列表构建待猜测的英文单词库列表 WORDS,使用 random 模块的 choice()函数从单词的元组中随机抽取一个英文单词 word,然后把该英文单词的字母乱序排列(方法是每次随机抽取一个位置的字符放入乱序的 jumble 字符串中,并从原 word 中删除

该字符)。

游戏一开始先显示乱序后的字符串 jumble,并提示用户输入猜测的结果,如果错误,将提示继续输入,直到输入正确。猜对之后可以询问是否继续游戏。游戏也可以通过 Ctrl+C 组合键强制中断运行。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

第 6 章

输入和输出



视频讲解

Python 程序通常包括输入和输出，以实现程序与外部世界的交互。

6.1 输入和输出概述

程序通过输入接收待处理的数据，然后执行相应的处理，最后通过输出返回处理的结果。其示意图如图 6-1 所示。

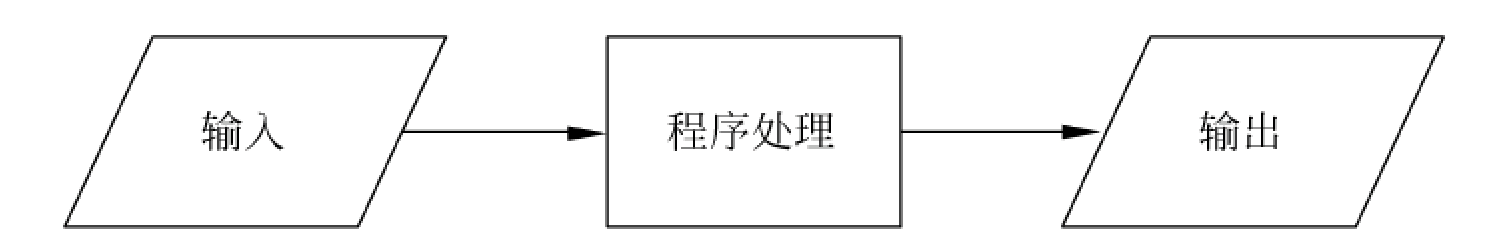


图 6-1 程序的输入和输出示意图

Python 程序通常可以使用下列方式之一实现交互功能。

(1) 命令行参数。

(2) 标准输入和输出函数。

(3) 文件输入和输出。

(4) 图形化用户界面(参见第 12 章)。

6.2 命令行参数

6.2.1 sys.argv 与命令行参数

命令行参数是 Python 语言的标准组成，是用户在命令行中 Python 程序之后输入的参数，在程序中可以通过 sys.argv 访问命令行参数。argv[0]为 Python 脚本名，argv[1]为第一个参数，argv[2]为第二个参数，依此类推。

按惯例，命令行输入参数 argv[1]、argv[2]等为字符串，所以如果希望传入的参数为数值，则需要使用转换函数 int()或 float()，将字符串转换为适合的类型。

【例 6.1】 命令行参数示例(randomseq.py)：生成 n 个随机数，其中 n 由程序的第一个命令行参数确定。

```
import sys, random
n = int(sys.argv[1])
for i in range(n):
    print(random.randrange(0,100))
```

程序运行结果如图 6-2 所示。

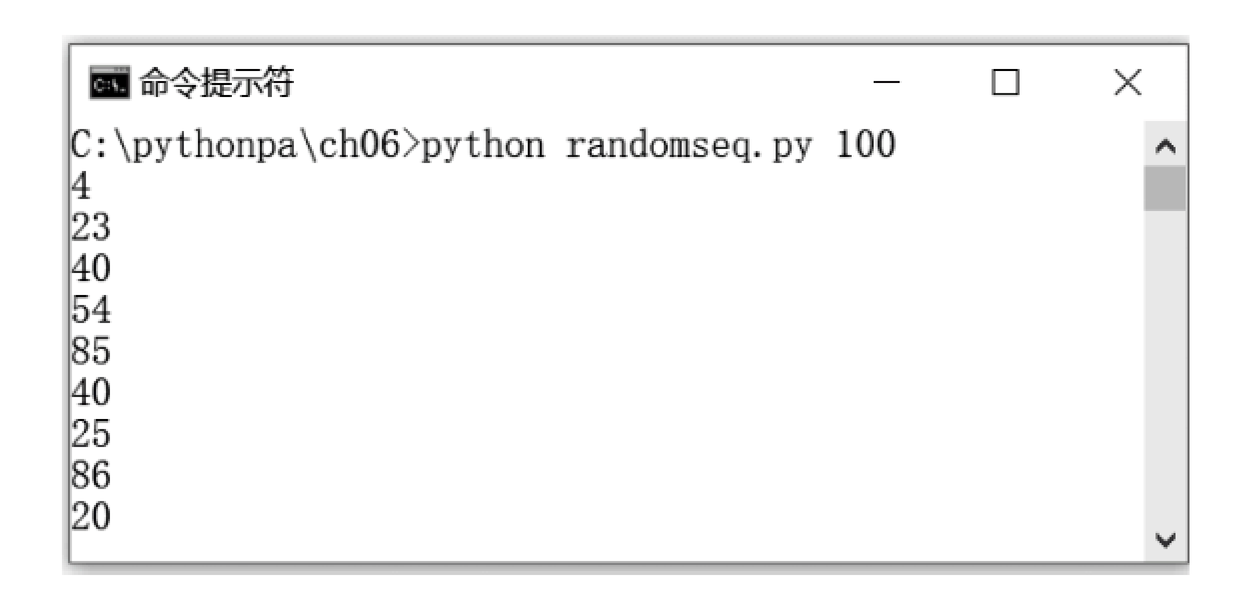


图 6-2　命令行参数确认随机数的个数

6.2.2　argparse 模块和命令行参数解析

argparse 模块是用于解析命名的命令行参数，生成帮助信息的 Python 标准模块。使用 argparse 模块的基本步骤如下。

（1）导入模块。

```
>>> import argparse
```

（2）创建 ArgumentParser 对象。

```
>>> parser = argparse.ArgumentParser()
```

（3）调用 parser 对象方法 add_argument()增加要解析的命令参数信息。

```
>>> parser.add_argument('--length', default = 10, type = int, help = '长度')
```

（4）调用 parser 对象方法 parse_args()解析命令行参数，生成对应的列表。

```
>>> args = parser.parse_args()
>>> args              #输出:Namespace(length=10)
```

【例 6.2】　命令行参数解析示例(arg_parse.py)：解析命令行参数所输入的长和宽的值，计算并输出长方形的面积。

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--length', default=10, type=int, help='长度')
parser.add_argument('--width', default=5, type=int, help='宽度')
args=parser.parse_args()
area=args.length * args.width
print('面积=', area)
```

程序运行结果如图 6-3 所示。

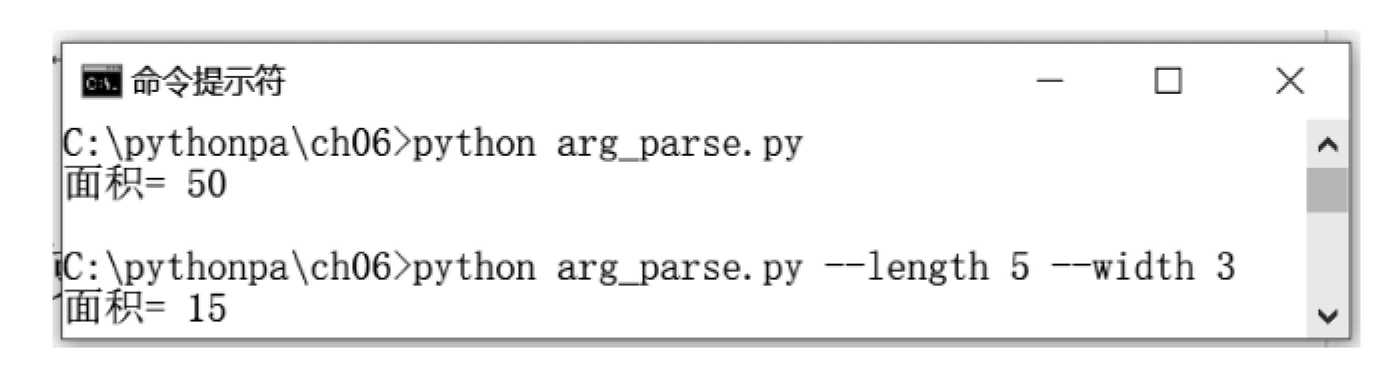


图 6-3　命令行参数确认长方形的长和宽

6.3 标准输入和标准输出函数

6.3.1 输入和输出函数

通过 Python 内置的输入函数 input()和输出函数 print()可以使程序与用户进行交互。

input()函数的格式如下。

```
input([prompt])
```

input()函数提示用户输入，并返回用户从控制台输入的内容（字符串）。

print()函数的格式如下。

```
print(value, …, sep = ' ', end = '\n', file = sys.stdout, flush = False)
```

print()函数用于打印一行内容，即将多个以分隔符（sep，默认为空格）分隔的值（value，…，以逗号分隔的值）写入到指定文件流（file，默认为控制台 sys. stdout）。其中，参数 end 指定换行符，flush 指定是否强制写入到流。

【例 6.3】 输入函数和输出函数示例 1（io_test1. py）。

```
>>> print(1,2,3)                        #输出时采用默认分隔符(空格).输出:1 2 3
>>> print(1,2,3,sep = ',')              #输出时采用逗号(,)分隔符.输出:1,2,3
>>> print(1,2,3,sep = ',',end = '.\n')  #输出时采用逗号分隔符,最后以点结束并换行
>>> for i in range(5):                  #输出时使用空格代替换行符
        print(i, end = ' ')
0 1 2 3 4
```

【例 6.4】 输入函数和输出函数示例 2（io_test2. py）。

```
import datetime
sName = input("请输入您的姓名:")                #输入姓名
birthyear = int(input("请输入您的出生年份:"))  #输入出生年份
age = datetime.date.today().year - birthyear  #根据当前年份和出生年份计算年龄
print("您好!{0}。您{1}岁。".format(sName, age))
```

程序运行结果如下。

```
请输入您的姓名:张三
请输入您的出生年份:1990
您好!张三。您 28 岁。
```

【例 6.5】 从控制台读取 n 个整数并计算其累计和（io_sum. py）。其中，n 由程序的第一个命令行参数所确定。

```
import sys
n = int(sys.argv[1])                      #命令行的第一个参数确认所需求和的整数个数 n
sum = 0                                   #设置求和初始值 = 0
for i in range(n):
    number = int(input('请输入整数:'))     #输入整数
    sum += number                         #整数累加
print('累计和为:', sum)                    #输出 n 个整数的累计和
```

程序运行结果如图 6-4 所示。

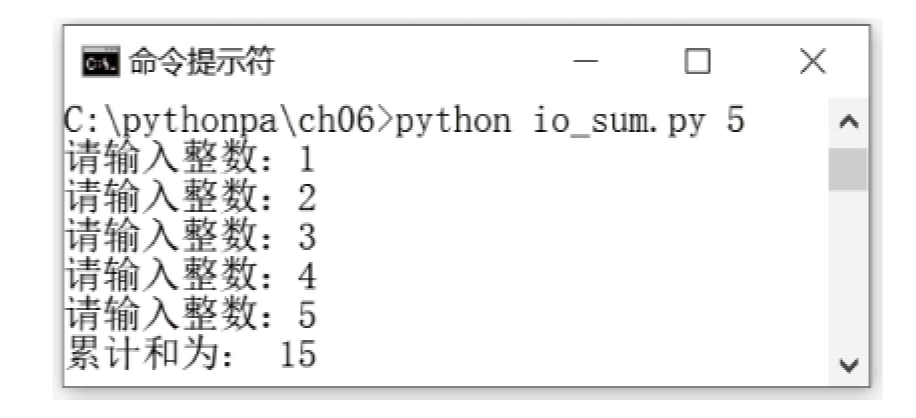


图 6-4 命令行参数确认所需求和的整数个数

6.3.2 交互式用户输入

在编写控制台应用程序时经常需要实现交互式用户输入，根据用户输入可以在程序执行过程中改变其控制流程。

编写支持用户交互的程序必须考虑各种可能的用户输入，因此会导致程序的复杂度提高。注意，现代程序一般使用图形用户界面接收用户输入。

【例 6.6】 编写程序(stat.py)，输入批量数据(假定当输入−1 时中止输入)，统计所输入的数据个数，并求总和以及平均值。

```
a = []                                          # 初始化列表
x = float(input("请输入一个实数,输入 - 1 中止:"))
while x != - 1:
    a.append(x)                                 # 将所输入的实数添加到列表中
    x = float(input("请输入一个实数,输入 - 1 中止:"))
print("计数:", len(a))                          # 列表长度即为实数个数
print("求和:", sum(a))                          # 列表中的各元素求和
print("平均值:", sum(a)/len(a))                 # 列表中的各元素求平均值
```

程序运行结果如下。

```
请输入一个实数,输入 - 1 中止:1.5
请输入一个实数,输入 - 1 中止:2.8
请输入一个实数,输入 - 1 中止:4.5
请输入一个实数,输入 - 1 中止:3.89
请输入一个实数,输入 - 1 中止:56.78
请输入一个实数,输入 - 1 中止: - 1
计数: 5
求和: 69.47
平均值: 13.894
```

6.3.3 运行时提示输入密码

如果在程序运行时需要提示用户输入密码，则可以使用模块 getpass，以保证用户输入的密码在控制台中不回显。在 getpass 模块中包含以下两个函数：

```
getpass.getpass(prompt = 'Password: ', stream = None)    # 提示用户输入密码并返回
getpass.getuser()                                        # 获取当前登录用户名
```

如果系统不支持不回显，则 getpass 模块将导致 getpass.GetPassWarning。

【例 6.7】 运行时提示输入密码(getpass1.py)。

```
import getpass
username = input("用户名:")                    # 提示输入用户名
passwd = getpass.getpass("密码:")              # 提示输入密码
```

```
if username == 'jianghong' and passwd == 'password':          #在实际运用中需要与数据库中的账户信
                                                              #息比较
    print('登录成功')
else:
    print('登录失败')
```

程序运行结果如图 6-5 所示。注意,该程序在集成开发环境 IDLE 下按 F5 键(或者执行 IDLE 菜单命令 Run|Run Module)运行时会产生安全问题,将导致运行失败。

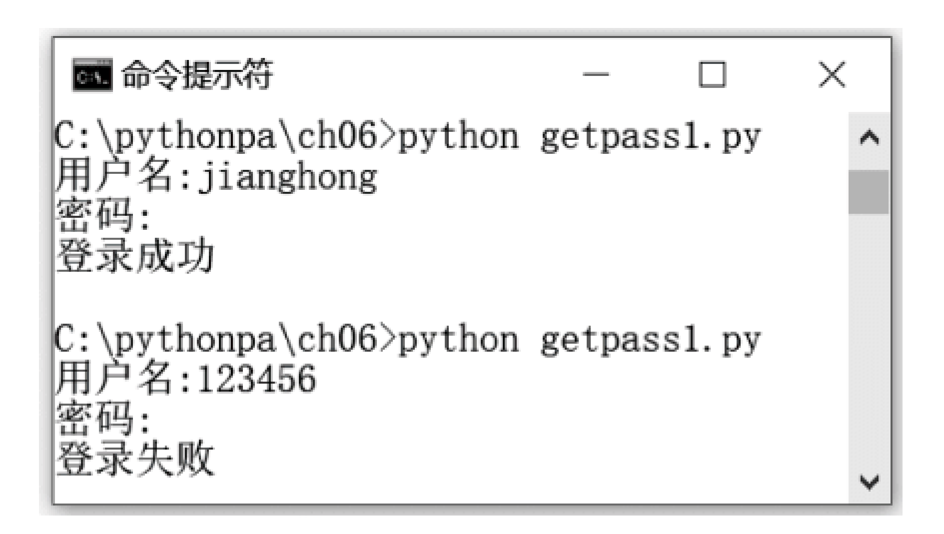


图 6-5 运行时提示输入用户名和密码

6.4 文件和文件对象

文件可以看作是数据的集合,一般保存在磁盘或其他存储介质上。

6.4.1 文件对象和 open()函数

内置函数 open()用于打开或创建文件对象,其语法格式如下。

```
f = open(file, mode='r', buffering=-1, encoding=None)
```

其中,file 是要打开或创建的文件名,如果文件不在当前路径,需指出具体路径;mode 是打开文件的模式;buffering 表示是否使用缓存(默认为−1,表示使用系统默认的缓冲区大小);encoding 是文件的编码。open()函数返回一个文件对象 f。

在使用 open()函数时可以指定打开文件的模式为'r'(只读)、'w'(写入,写入前删除旧内容)、'x'(创建新文件,如果文件存在,则导致 FileExistsError)、'a'(追加)、'b'(二进制文件)、't'(文本文件,默认值)、'+'(更新,读写)。

open()函数默认打开模式为'rt',即文本读取模式。

文件操作容易产生异常,而且最后需要关闭打开的文件,故一般使用 try…except…finally 语句,在 try 语句块中执行文件的相关操作,使用 except 捕获可能发生的异常,在 finally 语句块中确保关闭打开的文件。

```
try:
    f = open(file, mode)                          #打开文件
    #操作打开的文件
except:                                           #捕获异常
    #发生异常时执行的操作
finally:
    f.close()                                     #关闭打开的文件
```

6.4.2 文件的打开、写入、读取和关闭

通过内置函数 open()可以创建或打开文件对象;通过文件对象的实例方法 write/writelines 可以写入字符串到文本文件;通过文件对象的实例方法 read/readline 可以读取文本文件的内容;文件读写完成后,应该使用 close 方法关闭文件。

文本文件对象是可迭代对象,也可以使用 for 循环语句遍历所有的行。

【例 6.8】 读取并输出文本文件(type_file.py)。

```
import sys
filename = sys.argv[0]                    #所读取并输出的就是本程序文件 type_file.py
f = open(filename, 'r', encoding = 'utf8')  #打开文件
line_no = 0                               #统计行号
while True:
    line_no += 1                          #行号计数
    line = f.readline()                   #读取行信息
    if line:
        print(line_no, ":", line)         #输出行号和该行内容
    else:
        break
f.close()                                 #关闭打开的文件
```

程序运行结果如图 6-6 所示。

```
1 : import sys
2 : filename = sys.argv[0]    #所读取并输出的就是本程序文件type_file.py
3 : f=open(filename, 'r', encoding='utf8')    #打开文件
4 : line_no=0                                 #统计行号
5 : while True:
6 :     line_no += 1          #行号计数
7 :     line = f.readline()   #读取行信息
8 :     if line:
9 :         print(line_no, ":", line)   #输出行号和该行内容
10 :     else:
11 :         break
12 : f.close()              #关闭打开的文件
```

图 6-6　读取并输出文本文件

6.4.3　with 语句和上下文管理协议

使用 try…except…finally 语句可以确保在 try 语句块中获得的资源(例如打开的文件)在 finally 语句块中释放。

为了简化操作,Python 语言中与资源相关的对象可以实现上下文管理协议。实现上下文管理协议的对象可以使用 with 语句:

```
with context [as var]
    操作语句
```

with 语句定义了一个上下文。在执行 with 语句时,首先调用上下文对象 context 的__enter__(),其返回值赋给 var;离开 with 语句块时,最后调用 context 的__exit__(),确保释放资源。

文件对象支持使用 with 语句,确保打开的文件自动关闭:

```
with open(file, mode) as f:
    #操作打开的文件
```

【例 6.9】　利用 with 语句读取并输出文本文件(type_file_with.py)。

```
import sys
filename = sys.argv[0]                    #所读取并输出的就是本程序文件 type_file_with.py
```

```
line_no = 0                                            #统计行号
with open(filename, 'r', encoding = 'utf8') as f:      #使用 with 语句实现上下文管理协议
    for line in f:
        line_no += 1                                   #行号计数
        print(line_no, ":", line)                      #输出行号和该行内容
f.close()
```

6.5 标准输入、输出和错误流

6.5.1 标准输入、输出和错误流文件对象

在程序启动时,Python 自动创建并打开 3 个文件流对象,即标准输入流文件对象、标准输出流文件对象和错误输出流文件对象。

使用 sys 模块的 sys. stdin、sys. stdout 和 sys. stderr 可以查看对应的标准输入、标准输出和标准错误流文件对象。

```
>>> import sys
>>> sys.stdin       #输出:<_io.TextIOWrapper name = '<stdin>' mode = 'r' encoding = 'utf-8'>
>>> sys.stdout      #输出:<_io.TextIOWrapper name = '<stdout>' mode = 'w' encoding = 'utf-8'>
>>> sys.stderr      #输出:<_io.TextIOWrapper name = '<stderr>' mode = 'w' encoding = 'utf-8'>
```

标准输入流文件对象默认对应于控制台键盘。标准输出流文件对象和错误输出流文件对象默认对应于控制台,其区别仅在于后者是非缓冲的。

sys. stdout 的对象方法 write()用于输出对象的字符串表示到标准输出。事实上,print()函数就是调用 sys. stdout. write()方法。

【例 6.10】 标准输出流示例。

```
>>> import sys
>>> print("An error message", file = sys.stdout)       #输出:An error message
>>> sys.stdout.write("Another error message\n")
Another error message
22
```

6.5.2 读取任意长度的输入流

程序可以从输入流(sys. stdin)中读取数据直到输入流为空。理论上,输入流的大小没有限制。现代操作系统通常使用组合键 Ctrl+D 指示输入流结束(也有操作系统使用组合键 Ctrl+Z,例如 Windows 操作系统)。

与使用命令行参数相比,标准输入允许用户与程序进行交互(使用命令行参数时只能在程序运行前为程序提供数据),且可以读取大量数据(使用命令行参数时有长度限制)。

使用标准输入还可以通过操作系统重定向标准输入的源(例如文件或其他程序的输出),从而实现输入的灵活性。

【例 6.11】 计算输入流中数值的平均值(average. py)。

```
import sys
total = 0.0
count = 0
for line in sys.stdin:
    count += 1
```

```
        total += float(line)
avg = total / count
print("平均值为:",avg)
```

程序运行结果如图 6-7 所示。

6.5.3　标准输入、输出和错误流重定向

通过设置 sys.stdin、sys.stdout 和 sys.stderr 可以实现标准输入、输出和错误流的重定向。例如：

```
f = open('out.log', 'w')
sys.stdout = f
```

恢复标准输入、输出和错误流为默认值的代码如下。

```
sys.stdin = sys.__stdin__
sys.stdout = sys.__stdout__
sys.stderr = sys.__stderr__
```

【例 6.12】　标准输出流重定向示例(poweroftwo.py)。从命令行的第一个参数中获取 n 的值，然后将 0～n 以及 2 的 0～n 次幂的列表打印输出到 out.log 文件中。

```
import sys
n = int(sys.argv[1])                    #从命令行的第一个参数中获取 n 的值
power = 1                               #2 的 0～n 次幂赋初值
i = 0                                   #计数赋初值
f = open('out.log', 'w')                #指定标准输出重定向到 out.log 文件中
sys.stdout = f
while i <= n:
    print(str(i), ' ', str(power))      #输出 0～n 以及 2 的 0～n 次幂的列表
    power = 2 * power                   #计算 2 的 0～n 次幂
    i = i + 1                           #计数加 1
sys.stdout = sys.__stdout__
print('done!')
```

程序运行结果如图 6-8 所示。

```
命令提示符
C:\pythonpa\ch06>python average.py
1
2
3
4
5
6
^Z
平均值为:  3.5
```

图 6-7　计算输入流中数值的平均值

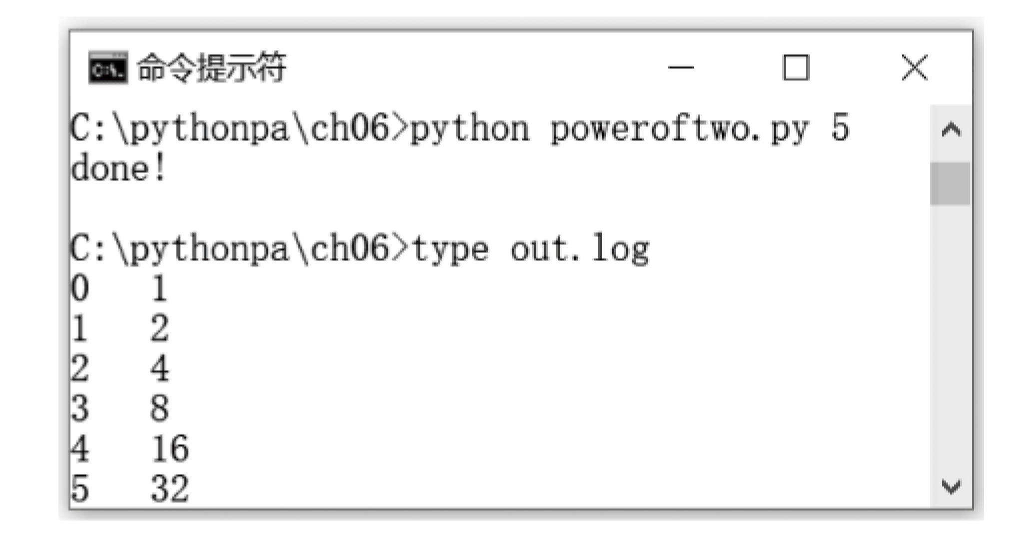


图 6-8　标准输出流重定向程序运行结果

6.6　重定向和管道

标准输入和标准输出对应于输入流和输出流。在默认情况下，键盘是标准输入流，显示屏是标准输出流。因此，在默认情况下，标准输入来自键盘的输入，而将标准输出结果发送到显示屏。

然而通过控制台键盘输入数据不适合于大量数据的情况，且每次运行都需要重新输入数据。现代操作系统都提供了标准输入和输出的重定向功能，把标准输入和标准输出关联的默认设备改变为另一文件、一个网络、一个程序等。

通过重定向可以实现标准输入和标准输出的抽象，并通过操作系统为标准输入或者标准输出指定不同的源。

6.6.1 重定向标准输出到一个文件

通过在执行程序的命令后面添加重定向指令可以将标准输出重定向到一个文件。程序将标准输出的结果写入指定文件，可以实现永久存储。

输出重定向的语法格式如下。

程序 > 输出文件

其目的是将显示屏从标准输出中分离，并将“输出文件”与标准输出关联，即“程序”的执行结果将写入“输出文件”，而不是发送到显示屏中显示。

【例 6.13】 重定向标准输出到一个文件的示例，如图 6-9 所示。

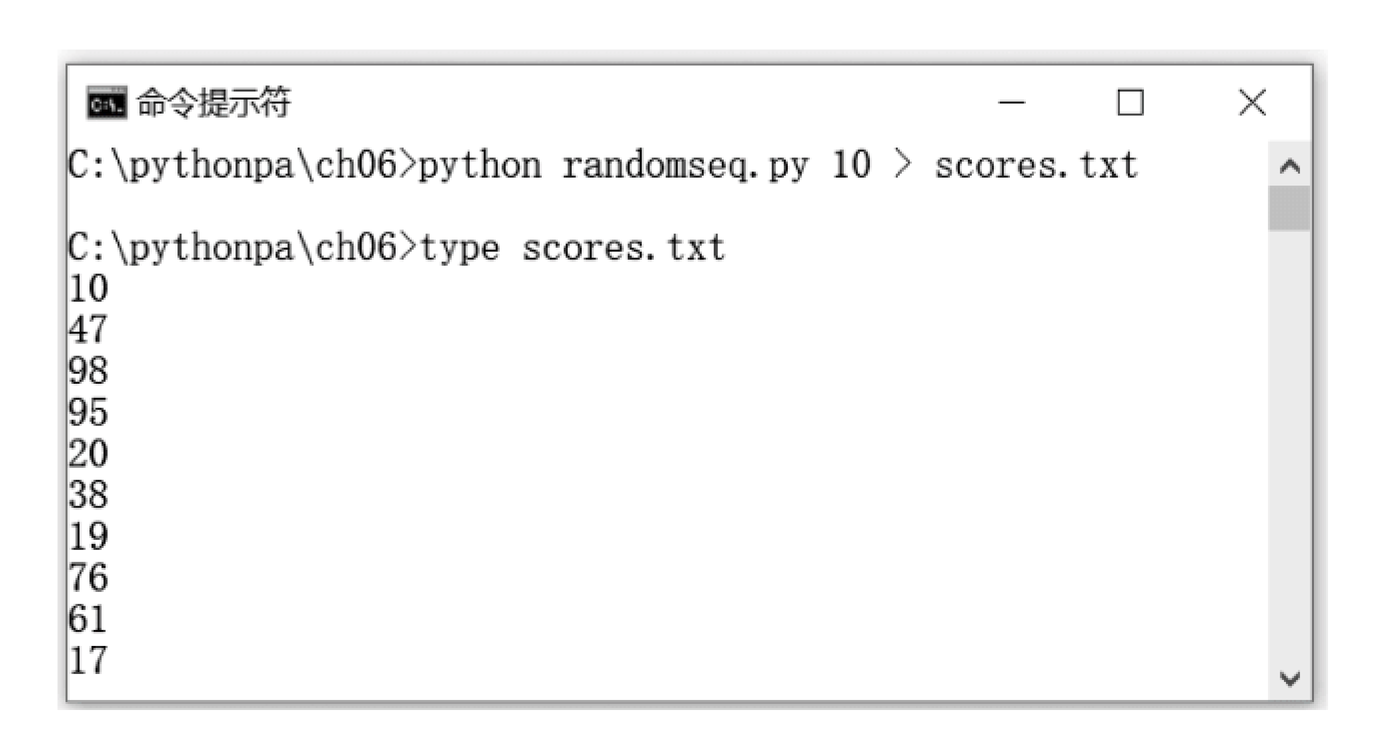


图 6-9 重定向标准输出到一个文件

重定向文件到标准输出的示意图如图 6-10 所示。

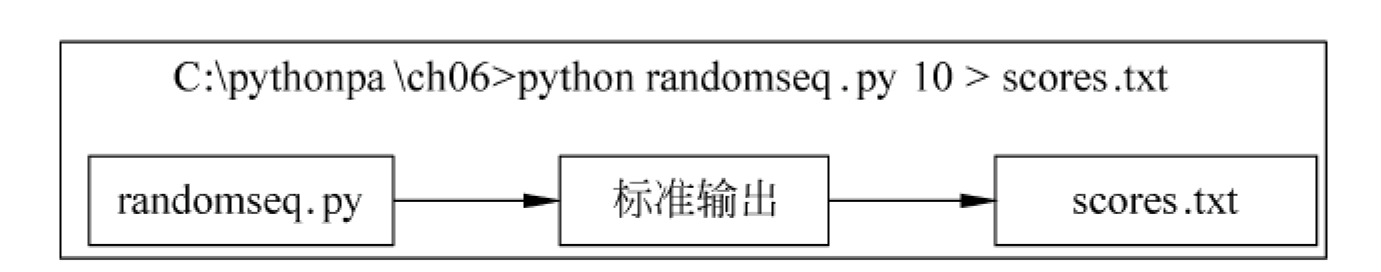


图 6-10 重定向文件到标准输出的示意图

6.6.2 重定向文件到标准输入

通过在执行程序的命令后面添加重定向指令可以实现程序从文件中读取输入数据，以代替从控制台程序中读取输入数据。

输入重定向的语法格式如下。

程序 < 输入文件

其目的是将控制台键盘从标准输入中分离，并将“输入文件”与标准输入流关联，即“程序”从“输入文件”中读取输入数据，而不是从键盘读取输入数据。

重定向文件到标准输入的功能可以实现“数据驱动的代码”，即不用修改程序就可以实

现处理不同数据文件。也就是将数据保存在文件中，通过编写程序从标准输入中读取数据。

【例 6.14】　重定向文件到标准输入的示例，如图 6-11 所示。

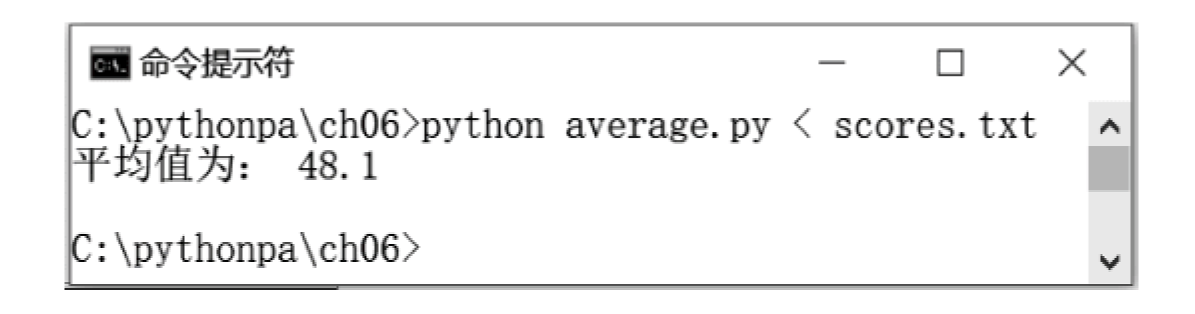


图 6-11　重定向文件到标准输入

重定向文件到标准输入的示意图如图 6-12 所示。

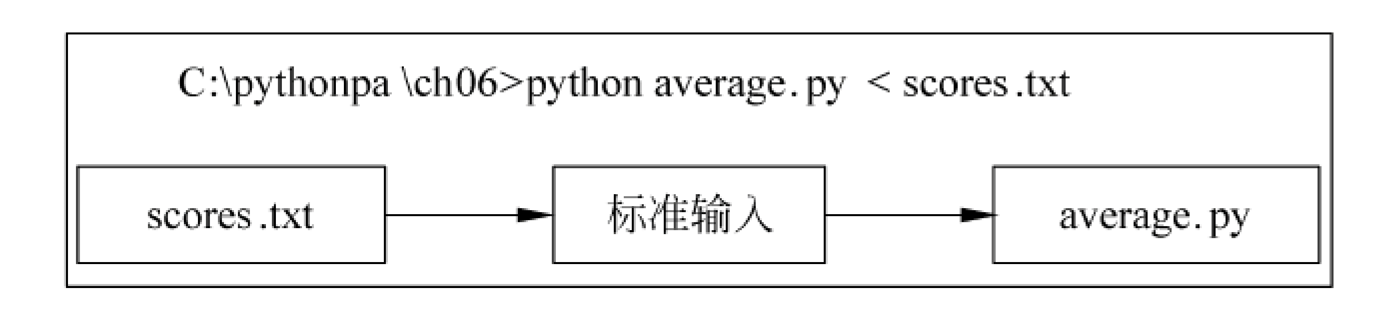


图 6-12　重定向文件到标准输入的示意图

6.6.3　管道

通过管道操作可以指定一个程序的输出为另一程序的输入，即将一个程序的标准输出与另一个程序的标准输入相连，这种机制称为管道。

管道操作的语法格式如下。

```
程序 1 | 程序 2 | … | 程序 n
```

其目的是将"程序 1"的标准输出连接到"程序 2"的标准输入，将"程序 2"的标准输出连接到"程序 3"的标准输入，依此类推。

例如：

```
C:\pythonpa\ch06 > python randomseq.py 1000 | python average.py
```

其执行结果等同于下列两行执行命令：

```
C:\pythonpa\ch06 > python randomseq.py 1000 > scores.txt
C:\pythonpa\ch06 > python average.py < scores.txt
```

当 randomseq.py 调用 print()函数时，一个字符串被添加到流的结尾；当 average.py 调用循环从 sys.stdin 读取数据时，一个字符串从流的头部被清除。

使用管道更加简洁，且不用创建中间文件，从而消除了输入流和输出流可以处理的数据大小的限制（例如，如果产生一万亿个随机数，则可能超出磁盘剩余空间的大小），执行效率更高。

管道执行的示意图如图 6-13 所示。

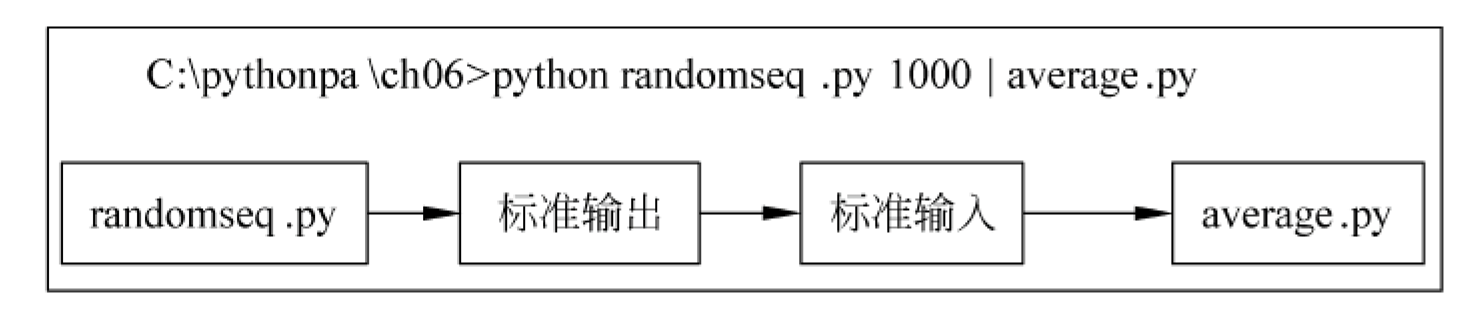


图 6-13　管道执行的示意图

6.6.4 过滤器

在现代操作系统中,使用管道作为命令行机制可以将多个程序串联起来。每个程序可以视为一个过滤器,过滤器通过某种形式将标准输入流转换为标准输出流。

【例 6.15】 过滤器示例 1:使用操作系统实用程序 more 逐屏显示数据,如图 6-14 所示。

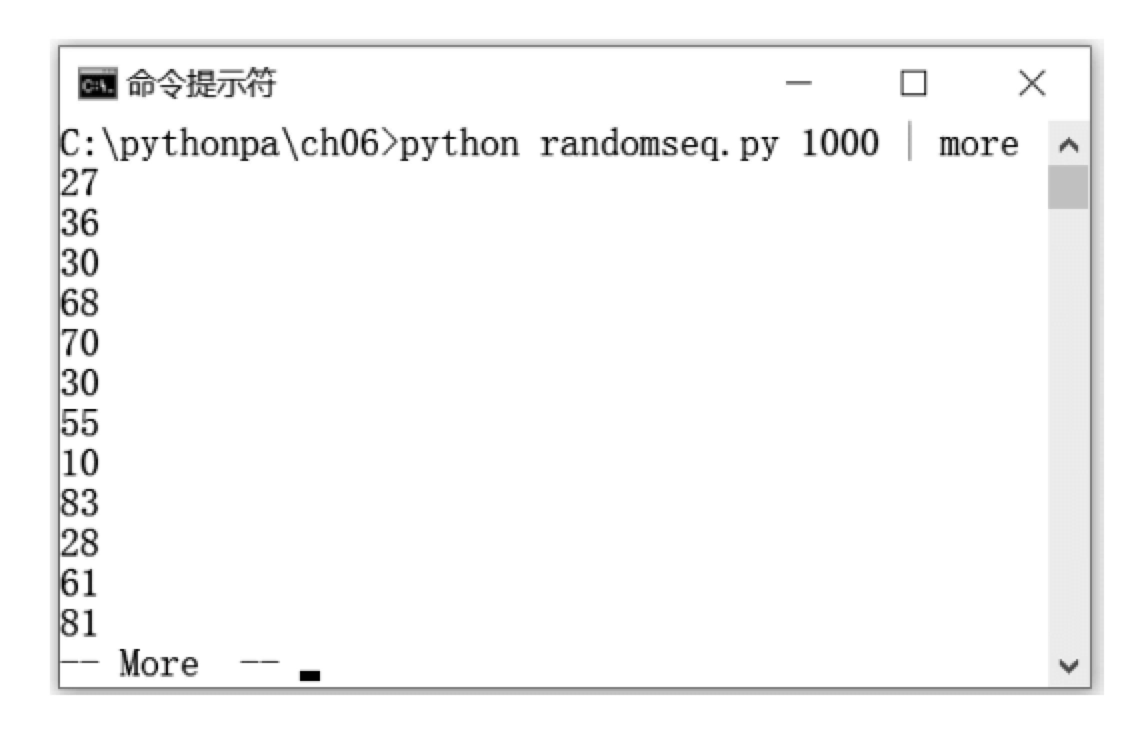


图 6-14 使用操作系统实用程序 more 逐屏显示数据

【例 6.16】 过滤器示例 2:使用操作系统实用程序 sort 排序输出数据,如图 6-15 所示。

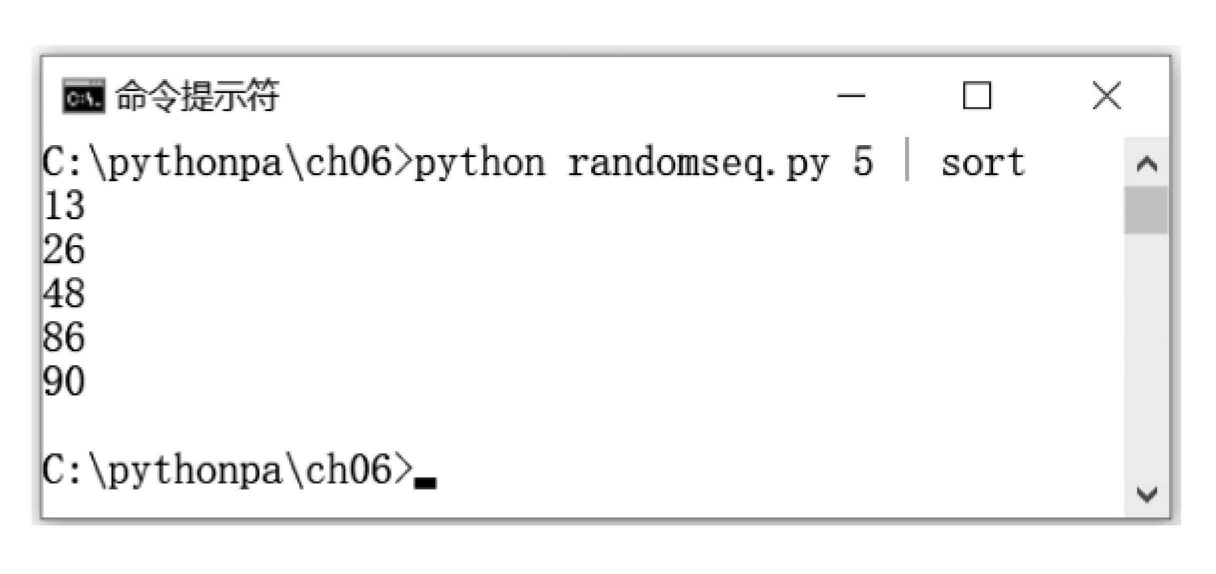


图 6-15 使用操作系统实用程序 sort 排序输出数据

【例 6.17】 过滤器示例 3:使用操作系统实用程序 sort 和 more 排序并逐屏输出数据,如图 6-16 所示。

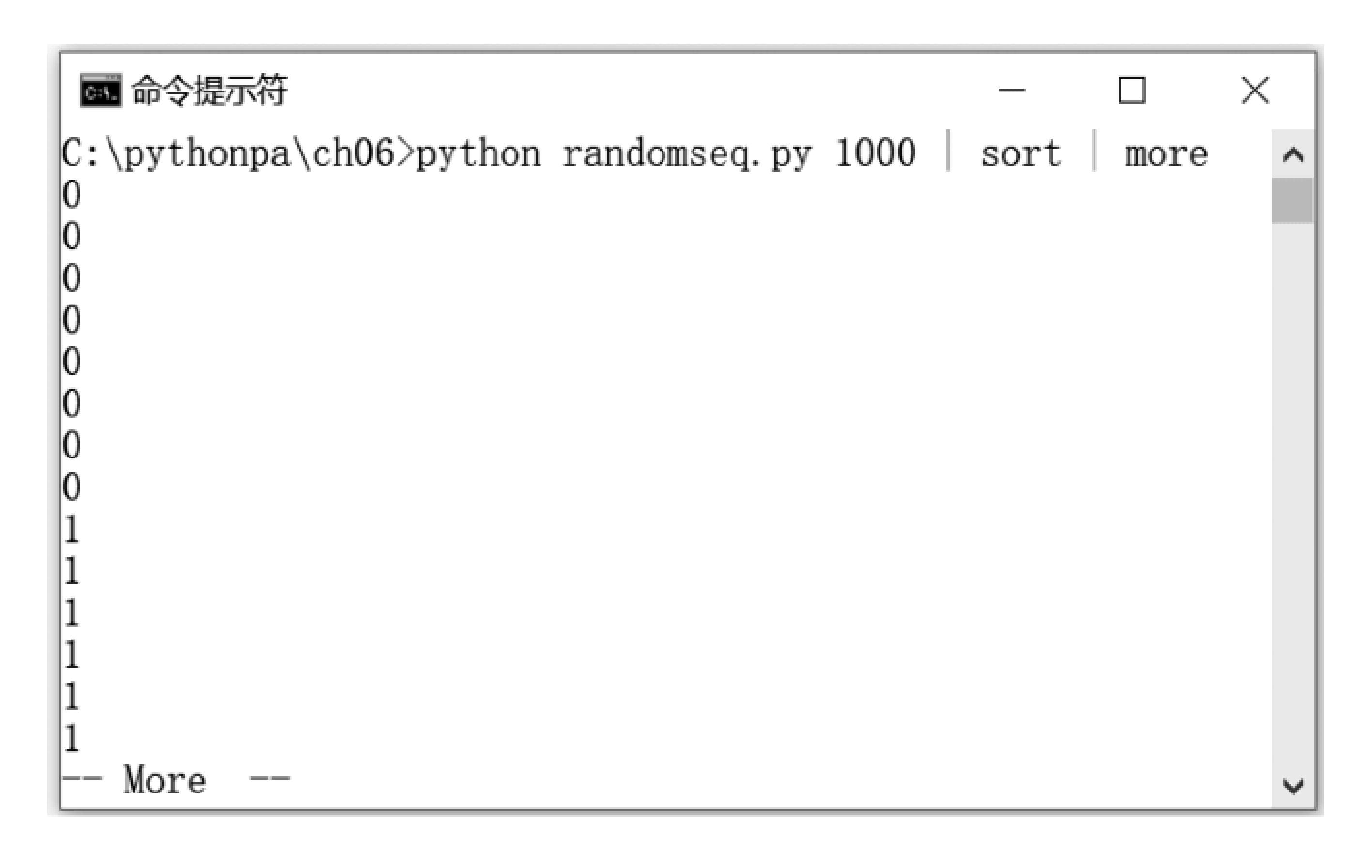


图 6-16 使用操作系统实用程序 sort 和 more 排序并逐屏输出数据

【例 6.18】 过滤器示例 4(rangefilter.py)：将来自标准输入中位于指定范围的值写入标准输出。

```
import sys
lo = int(sys.argv[1])
hi = int(sys.argv[2])
for line in sys.stdin:
    value = int(line)
    if (value >= lo) and (value <= hi):
        print(str(value))
```

程序运行结果如图 6-17 所示。

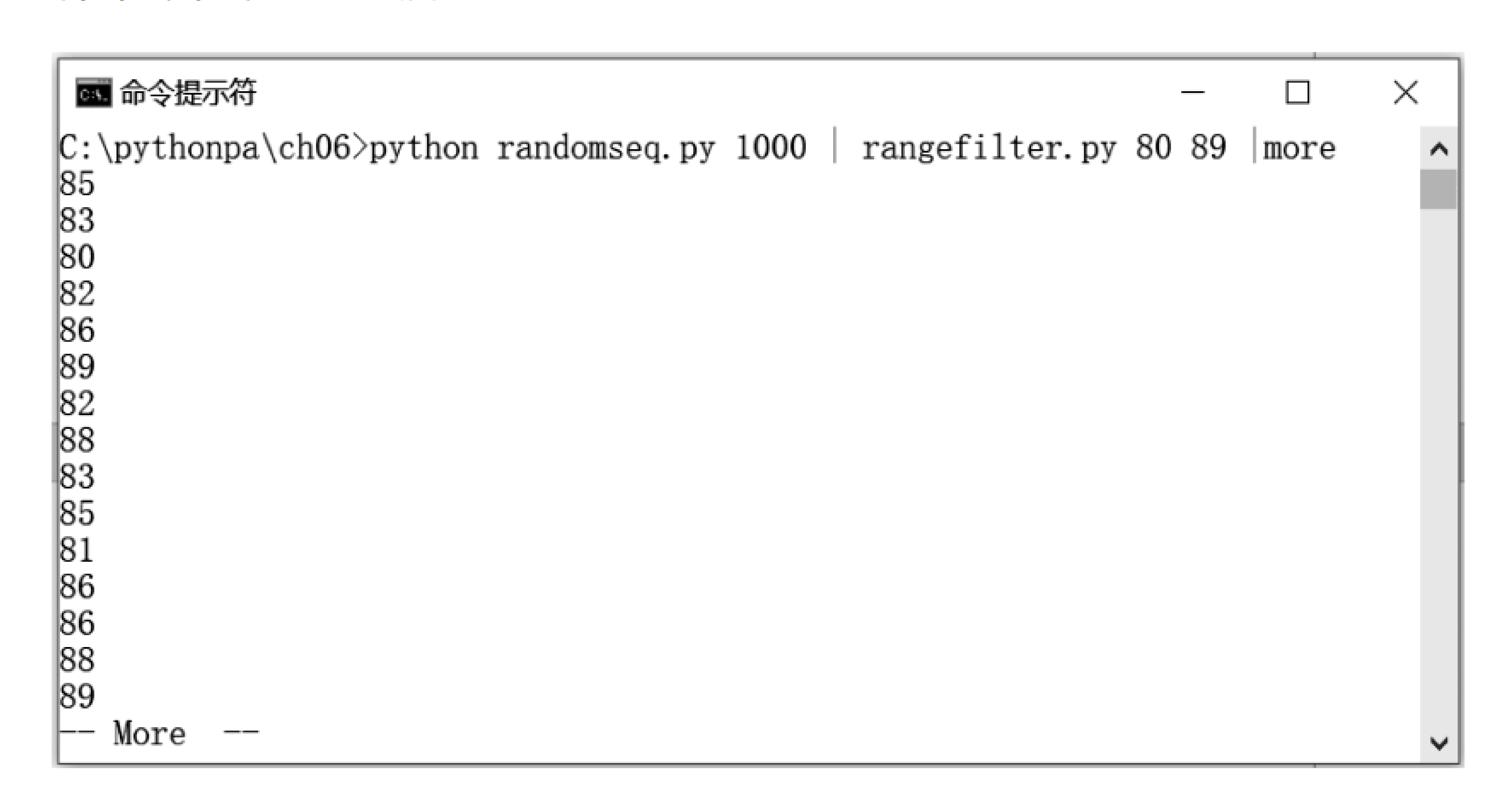


图 6-17　将来自标准输入中位于指定范围的值写入标准输出

6.7　复　习　题

一、填空题

1. Python 语句 print(1,2,3,4,5,sep='-',end='!')的输出结果是________。

2. Python 语句“for i in range(10): print(i, end=' ')”的输出结果是________。

3. 在 Python 程序中可以通过列表________访问命令行参数。________为 Python 脚本名，________为第一个参数名，________为第二个参数名。

4. Python 程序使用________模块解析命名的命令行参数。

5. 如果在程序运行时需要提示用户输入密码，则可以使用模块________，以保证用户输入的密码在控制台中不回显。

6. Python 语言使用________语句实现上下文管理协议。

7. 在 Python 语言中，使用 sys 模块中的________、________和________可以查看对应的标准输入、标准输出和标准错误流文件对象。

二、思考题

1. Python 程序通常可以使用哪几种方式实现交互功能？

2. 在 Python 中使用 argparse 模块解析命名的命令行参数的主要步骤是什么？

3. Python 内置的输入函数 input()的语法格式是什么？其具体的参数含义各是什么？

4. Python 内置的输出函数 print()的语法格式是什么？其具体的参数含义各是什么？

5. 在 Python 中使用 open()函数时,可以指定哪些打开文件的模式?默认的打开模式是什么?

6. 请问输入重定向和输出重定向的语法格式分别是什么?

7. 请问管道机制实现什么功能?管道操作的语法格式是什么?

6.8 上机实践

1. 完成本章中的例 6.1~例 6.18,熟悉 Python 程序的输入和输出功能。

2. 尝试修改例 6.2 编写命令行参数解析的程序,解析命令行参数所输入边长的值,计算并输出正方形的周长和面积。

3. 尝试修改例 6.8 编写读取并输出文本文件的程序,由命令行第一个参数确认所需输出的文本文件名。

4. 尝试修改例 6.9 编写利用 with 语句读取并输出文本文件的程序,由命令行第一个参数确认所需输出的文本文件名。

5. 尝试修改例 6.12 编写标准输出流重定向的程序,从命令行第一个参数中获取 n 的值,然后将 0~n、0~n 的 2 倍值、2 的 0~n 次幂的列表打印输出到 out.log 文件中。

6.9 案例研究:21 点扑克牌游戏

本章案例研究通过一个稍微复杂的游戏案例帮助读者进一步了解使用数据结构和算法实现基本的游戏人工智能,实现人机对话交互功能。

“21 点扑克牌游戏”是一种流行的扑克牌游戏。游戏者的目标是使手中的牌的点数之和不超过 21 点且尽量大。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

第 7 章

错误和异常处理



视频讲解

在程序的编写和运行过程中不可避免地会产生错误(bugs)和异常(exceptions),Python 语言采用结构化的异常处理机制捕获和处理异常。

7.1 程序的错误

Python 程序的错误通常可以分为 3 种类型,即语法错误、运行时错误和逻辑错误。

7.1.1 语法错误

Python 程序的语法错误是指其源代码中的拼写语法错误,这些错误导致 Python 编译器无法把 Python 源代码转换为字节码,故也称之为编译错误。当程序中包含语法错误时,编译器将显示 SyntaxError 错误信息。

通过分析编译器抛出的运行时错误信息,仔细分析相关位置的代码,可以定位并修改程序错误。

【例 7.1】 Python 语法错误示例(syntax_error.py)。

```
print("Good Luck!"
print("你今天的幸运随机数是:", random.choice(range(10)))
```

程序运行结果如图 7-1 所示。

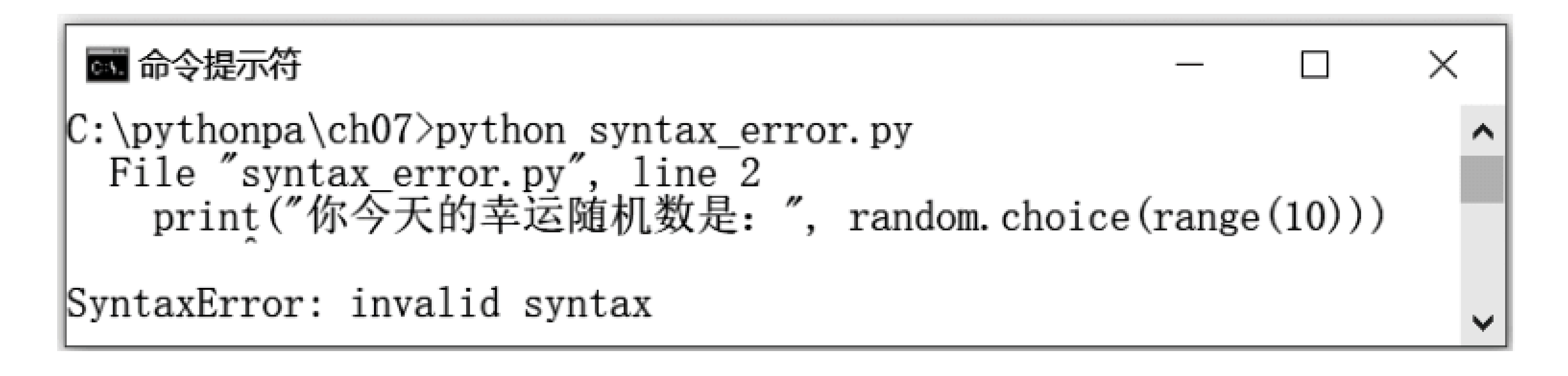


图 7-1 Python 语法错误运行示意图

编译器显示错误行号为 2,这是因为第一行的 print()函数需要结束括号,编译器编译到第二行时发现错误。在一般情况下,需要根据提示错误行号和信息在其附近判断和定位具体的错误。

7.1.2 运行时错误

Python 程序的运行时错误是在解释执行过程中产生的错误。例如,如果程序中没有导入相关的模块(例如 import random),解释器将在运行时抛出 NameError 错误信息;如果程序中包括零除运算,解释器将在运行时抛出 ZeroDivisionError 错误信息;如果程序中试图打开

不存在的文件，解释器将在运行时抛出 FileNotFoundError 错误信息。

通过分析解释器抛出的运行时错误信息，仔细分析相关位置的代码，可以定位并修改程序错误。

【例 7.2】 Python 运行时错误（没有导入相关的模块）示例（name_error.py）。

```
print("Good Luck!")
print("你今天的幸运随机数是:", random.choice(range(10)))
```

程序运行结果如图 7-2 所示。

```
命令提示符
C:\pythonpa\ch07>python name_error.py
Good Luck!
Traceback (most recent call last):
  File "name_error.py", line 2, in <module>
    print("你今天的幸运随机数是: ", random.choice(range(10)))
NameError: name 'random' is not defined
```

图 7-2 Python 运行时错误（没有导入相关的模块）示例运行示意图

编译器显示错误行号为 2，这是因为程序中没有导入相关模块的语句（import random），编译器编译到第二行时发现错误。在一般情况下，需要根据提示错误行号和信息在其附近判断和定位具体的错误。

【例 7.3】 Python 运行时错误（零除错误）示例（zero_division_error.py）。

```
a = 1
b = 0
c = a/b
```

程序运行结果如图 7-3 所示。

```
命令提示符
C:\pythonpa\ch07>python zero_division_error.py
Traceback (most recent call last):
  File "zero_division_error.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

图 7-3 Python 运行时错误（零除错误）示例运行示意图

7.1.3 逻辑错误

Python 程序的逻辑错误是程序可以执行（程序运行本身不报错），但执行结果不正确。对于逻辑错误，Python 解释器无能为力，需要用户根据结果来调试判断。

【例 7.4】 Python 逻辑错误示例（logic_error.py）。

```
import math
a = 1; b = 2; c = 1
x1 = -b + math.sqrt(b * b - 4 * a * c)/2 * a            #公式有误，故结果不正确
x2 = -b - math.sqrt(b * b - 4 * a * c)/2 * a            #公式有误，故结果不正确
print(x1, x2)                                           #输出：-2.0 -2.0
```

程序计算一元二次方程$ax^2+bx+c=0$ 的两个根：$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$。方程 x^2+2x+

1=0 的正确解为 x1=x2=−1。但由于计算公式有误(正确公式为(−b + math.sqrt(b * b−4 * a * c))/(2 * a)以及(−b − math.sqrt(b * b−4 * a * c))/(2 * a)),结果不正确。

7.2 异常处理

7.2.1 异常处理概述

Python 语言采用结构化的异常处理机制。在程序运行过程中,如果产生错误,将抛出异常;通过 try 语句来定义代码块,以运行可能抛出异常的代码;通过 except 语句可以捕获特定的异常并执行相应的处理;通过 finally 语句可以保证即使产生异常(处理失败),也可以在事后清理资源等。例如,读取文件内容的伪代码一般如下。

```
def readfile():
    打开文件                                    #可能产生错误:文件不存在
    读取文件内容                                #可能产生错误:无读取权限
    关闭文件
```

若使用 Python 的结构化异常处理机制,其伪代码一般如下。

```
def read_file():
    try:
        打开文件                                #可能产生错误:文件不存在
        读取文件内容                            #可能产生错误:无读取权限
        关闭文件
    except FileNotFoundError:                   #捕获异常:无法打开文件
        #异常处理逻辑
    except PermissionError:                     #捕获异常:无读取权限
        #异常处理逻辑
```

从上面的伪代码可以看出,异常处理机制可以把错误处理和正常代码逻辑分开,从而可以更加高效地实现错误处理,增加程序的可维护性。

异常处理机制已经成为许多现代程序设计语言处理错误的标准模式。

7.2.2 内置的异常类

在程序运行过程中,如果出现错误,Python 解释器会创建一个异常对象,并抛出给系统运行时。即程序中止正常执行流程,转而执行异常处理流程。

在某种特殊条件下,代码中也可以创建一个异常对象,并通过 raise 语句抛出给系统运行时。

异常对象是异常类的对象实例。Python 异常类均派生于 BaseException,Python 内置的异常类的层次结构如图 7-4 所示。

【例 7.5】 常见异常示例。

(1) NameError:尝试访问一个未声明的变量。

```
>>> noname             #报错.NameError: name 'noname' is not defined
```

(2) SyntaxError:语法错误。

```
>>> int a              #报错.SyntaxError: invalid syntax
```

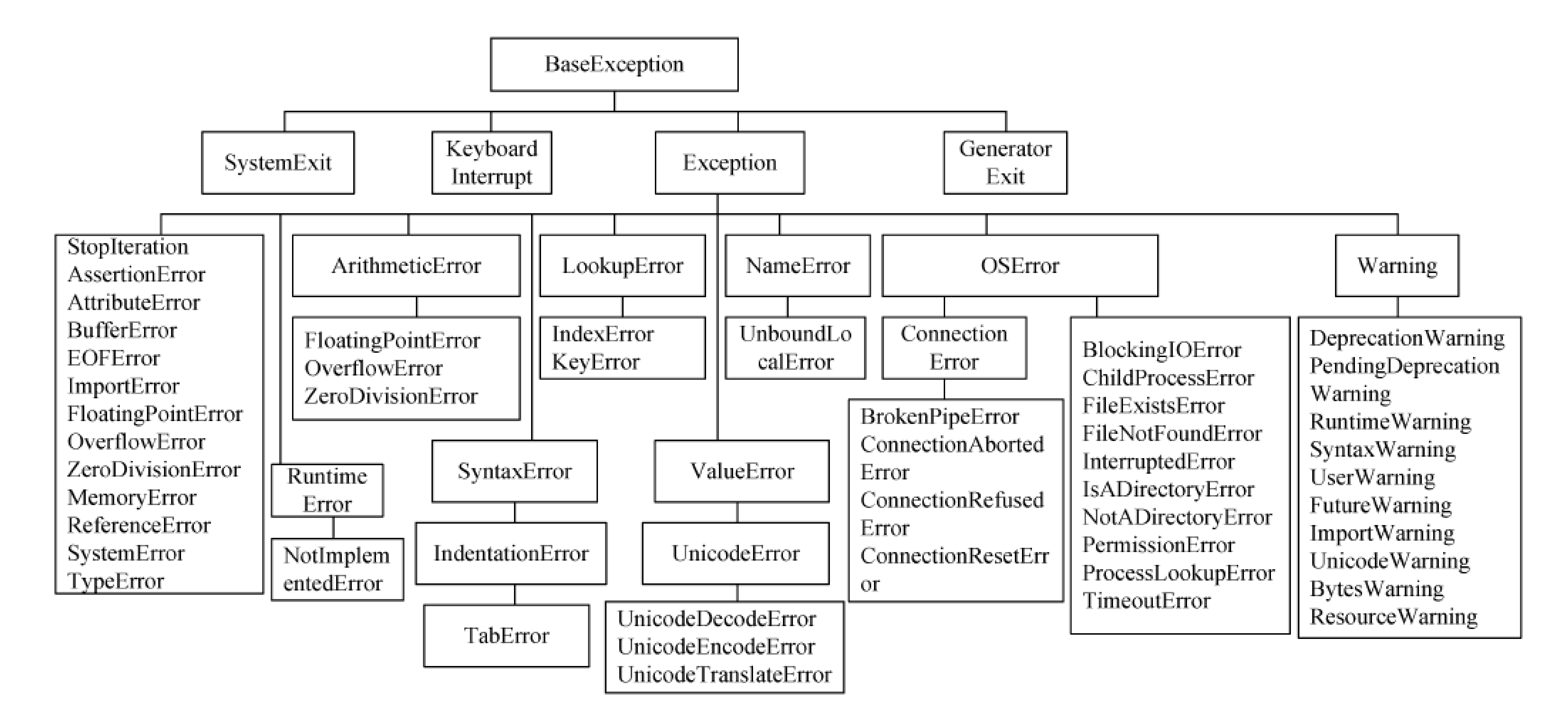


图 7-4　Python 异常类的层次结构

（3）AttributeError：访问未知对象属性。

```
>>> a = 1
>>> a.show()          # 报错.AttributeError: 'int' object has no attribute 'show'
```

（4）TypeError：类型错误。

```
>>> 11 + 'abc'        # 报错.TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

（5）ValueError：数值错误。

```
>>> int('abc')        # 报错.ValueError: invalid literal for int() with base 10: 'abc'
```

（6）ZeroDivisionError：零除错误。

```
>>> 1/0               # 报错.ZeroDivisionError: division by zero
```

（7）IndexError：索引超出范围。

```
>>> a = [10,11,12]
>>> a[3]              # 报错.IndexError: list index out of range
```

（8）KeyError：字典关键字不存在。

```
>>> m = {'1':'yes', '2':'no'}
>>> m['3']            # 报错.KeyError: '3'
```

7.2.3　引发异常

大部分由程序错误而产生的错误和异常一般由 Python 虚拟机自动抛出。另外，在程序中如果判断某种错误情况，可以创建相应的异常类的对象，并通过 raise 语句抛出。

【例 7.6】　Python 虚拟机自动抛出异常示例。

```
>>> 1/0               # 报错.ZeroDivisionError: division by zero
```

【例 7.7】　通过 raise 语句抛出异常示例。

```
>>> if a < 0: raise ValueError("数值不能为负数")
```

程序运行后将显示“ValueError：数值不能为负数”报错信息。

7.2.4 捕获处理异常机制概述

在程序中的某个方法抛出异常后，Python 虚拟机通过调用堆栈查找相应的异常捕获程序。如果找到匹配的异常捕获程序(即调用堆栈中的某函数使用 try…except 语句捕获处理)，则执行相应的处理程序(try…except 语句中匹配的 except 语句块)。如果堆栈中没有匹配的异常捕获程序，则 Python 虚拟机捕获处理异常。

7.2.5 Python 虚拟机捕获处理异常

如果堆栈中没有匹配的异常捕获程序，则该异常最后会传递给 Python 虚拟机，Python 虚拟机通用异常处理程序在控制台打印出异常的错误信息和调用堆栈，并中止程序的执行。

【例 7.8】 Python 虚拟机捕获处理异常示例(pvmexcept.py)。

```
i1 = 1
i2 = 0
print(i1/i2)
```

程序运行后将显示"ZeroDivisionError: division by zero"报错信息。

7.2.6 使用 try…except…else…finally 语句捕获处理异常

Python 语言采用结构化的异常处理机制。在程序运行过程中，如果产生错误，则抛出异常；try 语句定义代码块，运行可能抛出异常的代码；except 语句捕获特定的异常并执行相应的处理；else 语句执行无异常时的处理；finally 语句保证即使产生异常(处理失败)，也可以在事后清理资源等。try…except…else…finally 语句的一般格式如下。

```
try:
    可能产生异常的语句
except Exception1:                      #捕获异常 Exception1
    发生异常时执行的语句
except (Exception2, Exception3):        #捕获异常 Exception2、Exception3
    发生异常时执行的语句
except Exception4 as e:                 #捕获异常 Exception4,其实例为 e
    发生异常时执行的语句
except:                                 #捕获其他所有异常
    发生异常时执行的语句
else:                                   #无异常
    无异常时执行的语句
finally:                                #不管发生异常与否都保证执行
    不管发生异常与否都保证执行的语句
```

try 语句有以下 3 种可能的形式。

(1) try…except…[else…]语句：一个 try 块后接一个或多个 except 块，可选 else 块。

(2) try…finally 语句：一个 try 块后接一个 finally 块。

(3) try…except…[else…]finally 语句：一个 try 块后接一个或多个 except 块，可选 else 块，后面再跟一个 finally 块。

使用 try…except…else…finally 语句还可以重新引发异常，即处理部分异常，然后使用 raise 语句重新引发异常，以便调用堆栈中的其他异常处理程序捕获并处理。

【例 7.9】 try…except…else…finally 示例(try_except.py)。

```
try:
    f = open("testfile.txt", "w")
    f.write("这是一个测试文件,用于测试异常!!")
    f1 = open("testfile1.txt", "r")  #报错:没有找到文件或读取文件失败
except IOError:
    print("没有找到文件或读取文件失败")
else:
    print("文件写入成功!")
finally:
    f.close()
```

7.2.7 捕获异常的顺序

except 块可以捕获并处理特定的异常类型(此类型称为“异常筛选器”),具有不同异常筛选器的多个 except 块可以串联在一起。系统自动由上而下匹配引发的异常：如果匹配(引发的异常为“异常筛选器”的类型或子类型),则执行该 except 块中的异常处理代码,否则继续匹配下一个 except 块。故用户需要将带有最具体的(即派生程度最高的)异常类的 except 块放在最前面。

【例 7.10】 异常类位置顺序示例(try_except2.py)：派生程度高的异常类 NumberError 放置在派生程度低的 Exception 的后面,导致程序永远无法捕获。

```
a = (44, 78, 90, -80, 55)
total = 0
try:
    for i in a:
        if i < 0: raise ValueError(str(i) + "为负数")
        total += i
    print('合计 = ', total)
except Exception:
    print('发生异常')
except ValueError:
    print('数值不能为负')
```

在该程序中 Exception 是所有派生类的父类,故会首先捕获并处理,输出“发生异常”的提示信息；而后续的异常 ValueError 不能被捕获。两者的顺序应该交换。

7.2.8 finally 块和发生异常后的处理

finally 块始终在执行完 try 和 except 块之后执行,而与是否引发异常或者是否找到与异常类型匹配的 except 块无关。finally 块用于清理在 try 块中执行的操作,例如释放其占有的资源(如文件流、数据库连接和图形句柄),而不用等待由运行库中的垃圾回收器来完成对象。

【例 7.11】 使用 finally 语句保证执行代码示例(try_finally.py)：将输入的字符串写入文本中,直到按 Q 键结束；按 Ctrl+C 组合键中断程序的运行；最后保证打开的文件正常关闭。

```
try:
    f = open('mytext.txt', 'w')          #打开要写入的文件
    while True:
        s = input('请输入字符串(按 Q 键结束):')
        if s.upper() == 'Q': break
        f.write(s + '\n')
```

```
except KeyboardInterrupt:
    print('程序中断!(Ctrl-C)')
finally:
    f.close()
```

7.2.9 自定义异常类

在 Python 库中提供了许多异常。在应用程序的开发过程中,有时候需要定义特定于应用程序的异常类,表示应用程序的一些错误类型。

自定义异常类一般继承于 Exception 或其子类。自定义异常类的名称一般以 Error 或 Exception 为后缀。

【例 7.12】 创建自定义异常(NumberError.py),处理应用程序中出现负数参数的异常(例如学生成绩处理类,不允许成绩为负数)。

```
class NumberError(Exception): #自定义异常类,继承于 Exception
    def __init__(self,data):
        Exception.__init__(self, data)
        self.data = data
    def __str__(self):                    #重载__str__()方法
        return self.data + ': 非法数值(<0)'
def total(data):
    total = 0
    for i in data:
        if i<0: raise NumberError(str(i))
        total += i
    return total
#测试代码
data1 = (44, 78, 90, 80, 55)
print('总计=', total(data1))
data2 = (44, 78, 90, -80, 55)
print('总计=', total(data2))
```

程序运行结果如下。

```
总计= 347
Traceback(most recent call last):
  File "C:\pythonpa\ch07\NumberError.py", line 17, in <module>
    print('总计=', total(data2))
  File "C:\pythonpa\ch07\NumberError.py", line 10, in total
    if i<0: raise NumberError(str(i))
NumberError: -80: 非法数值(<0)
```

7.3 断言处理

7.3.1 断言处理概述

用户在编写程序时,在调试阶段往往需要判断代码执行过程中变量的值等信息(例如对象是否为空、数值是否为 0 等)。

用户可以使用 print()函数打印输出结果,也可以通过断点跟踪调试查看变量,但使用断

言更加灵活、高效。断言一般用于下列情况。

(1) 前置条件断言:代码执行之前必须具备的特性。

(2) 后置条件断言:代码执行之后必须具备的特性。

(3) 前后不变断言:代码执行前后不能变化的特性。

断言的主要功能是帮助程序员调试程序,以保证程序运行的正确性。断言一般在开发调试阶段使用,即在调试模式时断言有效,在优化模式运行时自动忽略断言。

与之相对应,异常处理则是通过错误的抛出和捕获机制来保证程序的健壮性。异常处理贯穿于程序开发运行的各个阶段。

7.3.2 assert 语句和 AssertionError 类

使用关键字 assert 可以声明断言。断言的声明有下列两种形式:

```
assert <布尔表达式>                    #简单形式
assert <布尔表达式>, <字符串表达式>    #带参数的形式
```

其中,<布尔表达式>的结果为一个布尔值(True 或 False),<字符串表达式>是断言失败时输出的失败消息。在调试模式下,如果<布尔表达式>为假,则抛出 AssertionError 对象实例。

Python 解释器有两种运行模式,即调试模式和优化模式。通常为调试模式,内置只读变量__debug__为 True;当使用选项-O 运行时(即 python. exe -O)为优化模式,此时内置只读变量__debug__为 False。故两种形式的 assert 语句相当于:

```
if __debug__:
   if not testexpression: raise AssertionError
if __debug__:
   if not testexpression: raise AssertionError(data)
```

【例 7.13】 断言示例(assert. py)。

```
a = int(input("请输入整数 a:"))
b = int(input("请输入整数 b:"))
assert b != 0, '除数不能为 0'
c = a / b
print(a, '/', b, '=', c)
```

程序运行结果如下。

```
请输入整数 a:1
请输入整数 b:0
Traceback(most recent call last):
  File "C:\Pythonpa\ch07\assert.py", line 3, in <module>
    assert b != 0, '除数不能为 0'
AssertionError: 除数不能为 0
```

7.3.3 启用/禁用断言

通常 Python 运行在调试模式,程序中的断言语句可以帮助程序员调错。在正式运行时,使用运行选项-O 以优化模式运行来禁用断言,从而提高程序效率。

【例 7.14】 启用/禁用断言选项示例:分别在两种模式下运行例 7.13,运行效果如图 7-5 所示。


```
C:\pythonpa\ch07>python assert.py
请输入整数a: 4
请输入整数b: 0
Traceback (most recent call last):
  File "assert.py", line 3, in <module>
    assert b != 0, '除数不能为0'
AssertionError: 除数不能为0

C:\pythonpa\ch07>python -O assert.py
请输入整数a: 4
请输入整数b: 0
Traceback (most recent call last):
  File "assert.py", line 4, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

图 7-5　启用/禁用断言运行效果

7.4　程序的基本调试方法

在程序实际运行之前，查找和修正其错误的过程称为调试(debugging)。

7.4.1　语法错误的调试

对于编译错误，Python 解释器会直接抛出异常。用户可以根据输出的异常信息修改程序代码。例如：

```
>>> if 1 > 2                  #报错.SyntaxError: invalid syntax
>>> Print('abc')              #报错.NameError: name 'Print' is not defined
>>> x                         #报错.NameError: name 'x' is not defined
```

根据 Python 解释器抛出的异常分别判断产生异常的原因：第一条语句产生的 SyntaxError 表示语法错误(在 if 复合语句后没有冒号)；第二条语句产生的 NameError 表示名称错误(拼写错误，应该为 print)；第三条语句产生的 NameError 表示名称错误(未定义变量)。

7.4.2　运行时错误的调试

对于运行时错误，Python 解释器也会抛出异常，在代码中可以通过 try…except 语句捕获并处理。如果在程序中没有 try…except，则 Python 解释器将直接打印出异常信息。

【例 7.15】　运行时错误调试示例。

```
>>> f = open('abc.txt')       #文件或者目录不存在:FileNotFoundError
>>> a = 1;b = 0
>>> c = a/b                   #零除溢出:ZeroDivisionError
>>> 123 + 'abc'               #TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

根据 Python 解释器运行时抛出的异常分别判断产生异常的原因：第一条语句产生的 FileNotFoundError 表示打开不存在的文件；第二条语句产生的 ZeroDivisionError 表示零除错误；第三条语句产生的 TypeError 表示不同类型对象值不能相加。

7.4.3　逻辑错误的调试

逻辑错误的调试方法包括断点跟踪、输出信息等方法，有的集成开发环境(IDE)可以设置断点并查看变量等。

通过 print 语句输出程序运行过程中变量的值(跟踪信息)是观察和调试程序运行逻辑正确性的有效方法。

【例 7.16】 通过输出信息跟踪逻辑错误调试示例(factor.py):将命令行参数所输入的整数分解为素数之积。

```
import sys
n = int(sys.argv[1])
result = []
factor = 2
while factor * factor <= n:
    while (n % factor) == 0:
        n //= factor
        result.append(factor)
        print(n, factor)
    factor += 1
if n > 1:
    result.append(n)
print(result)
```

运行效果如图 7-6 所示。

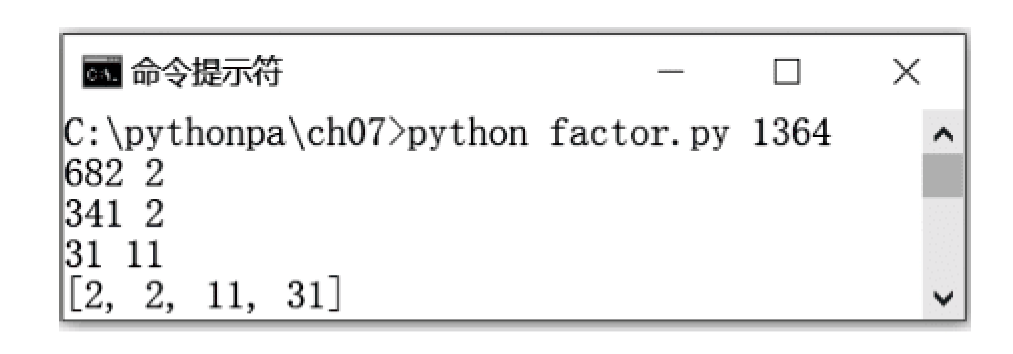


图 7-6 通过输出信息跟踪逻辑错误运行效果

7.5 使用 logging 模块输入日志

7.5.1 logging 模块概述

在运行或者调试程序时,记录日志有助于对错误的分析和处理,特别是服务器程序,记录日志十分必要。

虽然简单的程序开发和调试也可以使用 print 输出调试信息,但 print 会和正常的代码混在一起,混淆程序正常运行的逻辑。

在 Python 标准库中提供了用于记录日志的模块 logging。使用 Python 的标准日志模块可以在程序和库中灵活配置输出消息的级别,从而过滤掉并不重要的消息;可以配置输出消息的格式和内容;还可以配置日志输出的地方(例如控制台、文件等)。

logging 模块和 log4j 的机制是一致的,但具体的实现细节不同。logging 模块包括以下 4 个组成部分。

(1) logger:日志接口,用于配置和发送日志消息,可以通过 logging.getLogger(name)获取 logger 对象,如果不指定 name,则返回 root 对象。

(2) handler:日志处理器,将日志记录(log record)发送到目的地(destination,例如文件、socket 等)。一个 logger 对象可以通过 addHandler 方法添加零或者多个 handler,每个 handler 又可以定义不同日志级别,以实现日志分级过滤记录。

(3) filter：日志过滤器，确定一个日志记录是否发送到 handler。

(4) formatter：日志格式化，格式化要记录的日志。其构造方法包括两个可选参数，即消息的格式字符串和日期字符串。

7.5.2　logging 的配置和使用

使用 Python 代码配置 logging 的基本步骤如下。

(1) 创建 logger 对象。例如：

```
logger = logging.getLogger('log_test')          #获取名为 log_test 的 logger 对象
logger.setLevel(logging.DEBUG)                  #设置记录级别
#日志记录级别为 CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET
```

(2) 创建 handler 对象。例如：

```
fh = logging.FileHandler('log_test.log')        #创建文件处理器对象,记录详细信息
fh.setLevel(logging.DEBUG)
ch = logging.StreamHandler()                    #创建控制台处理器,输出错误信息
ch.setLevel(logging.ERROR)
```

(3) 创建 formatter 对象并和 handler 对象关联。例如：

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
```

其中，日期格式化主要包括以下参数。

- %(levelno)s：打印日志级别的数值。
- %(levelname)s：打印日志级别的名称。
- %(pathname)s：打印当前执行程序的路径，即 sys.argv[0]。
- %(filename)s：打印当前执行程序名。
- %(funcName)s：打印日志的当前函数。
- %(lineno)d：打印日志的当前行号。
- %(asctime)s：打印日志的时间。
- %(thread)d：打印线程 ID。
- %(threadName)s：打印线程名称。
- %(process)d：打印进程 ID。
- %(message)s：打印日志信息。

(4) 把 handler 对象添加到 logger 对象。例如：

```
logger.addHandler(fh)
logger.addHandler(ch)
```

(5) 在程序中使用 logger 对象的方法输出日志信息。例如：

```
logger.debug("调试信息")
logger.info("一般信息")
logger.warning("警告信息")
logger.error("错误信息")
logger.critical("严重信息")
```

除了上述配置方法以外，用户还可以使用 logging.basicConfig()便捷地配置日志。例如：

```
logging.basicConfig(level = logging.INFO,
                    format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
```

然后使用 logging 模块的函数输出日志信息。例如：

```
logging.debug("调试信息")
logging.info("一般信息")
logging.warning("警告信息")
logging.error("错误信息")
logging.critical("严重信息")
```

说明：最简单的日志记录方式直接使用上述 logging 模块的函数输出日志，即使用默认配置(level = logging.WARNING)，使用默认格式输出到控制台。

复杂的应用可以使用 logging.config.fileConfig("logging.conf")，从配置文件中读取配置。限于篇幅，本书对此不展开阐述，具体请读者查看 logging 模块的帮助。

【例 7.17】 使用默认配置直接输出到控制台(logging_default.py)。

```
import logging
logging.debug("调试信息") #不会输出
logging.info("一般信息")
logging.warning("警告信息")
logging.error("错误信息")
logging.critical("严重错误")
```

程序输出结果如下。

```
WARNING:root:警告信息
ERROR:root:错误信息
CRITICAL:root:严重错误
```

【例 7.18】 使用 basicConfig 配置输出日志到控制台(logging_console.py)。

```
import logging
#配置 logging
logging.basicConfig(level = logging.INFO,
               format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
#输出日志信息
logging.debug("调试信息") #不会输出
logging.info("一般信息")
logging.warning("警告信息")
logging.error("错误信息")
logging.critical("严重错误")
```

程序运行结果如下。

```
2018-06-25 17:00:56,546 - root - INFO - 一般信息
2018-06-25 17:00:56,566 - root - WARNING - 警告信息
2018-06-25 17:00:56,572 - root - ERROR - 错误信息
2018-06-25 17:00:56,577 - root - CRITICAL - 严重错误
```

【例 7.19】 使用 basicConfig 配置输出日志到文件(logging_file.py)。

```
import logging
#配置 logging
logging.basicConfig(filename="logging_file.txt", level = logging.INFO,
               format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
#输出日志信息
logging.debug("调试信息")
logging.info("一般信息")
```

```
logging.warning("警告信息")
logging.error("错误信息")
logging.critical("严重错误")
```

程序运行结果请参照生成的 logging_file.txt 文件，其中内容如下。

```
2018-06-25 17:01:46,103 - root - INFO - 一般信息
2018-06-25 17:01:46,107 - root - WARNING - 警告信息
2018-06-25 17:01:46,107 - root - ERROR - 错误信息
2018-06-25 17:01:46,107 - root - CRITICAL - 严重错误
```

【例 7.20】 输出日志信息到文件，同时输出错误信息到控制台(logging_console_file.py)。

```
import logging
#配置 logging
logger = logging.getLogger(__name__)
logger.setLevel(level = logging.DEBUG)
handler = logging.FileHandler("logging_console_file.txt")
handler.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)
console = logging.StreamHandler()
console.setLevel(logging.ERROR)
logger.addHandler(handler)
logger.addHandler(console)
#输出日志信息
logger.debug("调试信息")
logger.info("一般信息")
logger.warning("警告信息")
logger.error("错误信息")
logger.critical("严重错误")
```

程序运行结果如下。

```
错误信息
严重错误
```

同时生成日志文件 logging_console_file.txt，其内容如下。

```
2018-07-07 10:18:24,100 - __main__ - DEBUG - 调试信息
2018-07-07 10:18:24,101 - __main__ - INFO - 一般信息
2018-07-07 10:18:24,101 - __main__ - WARNING - 警告信息
2018-07-07 10:18:24,101 - __main__ - ERROR - 错误信息
2018-07-07 10:18:24,101 - __main__ - CRITICAL - 严重错误
```

7.6 复　习　题

一、选择题

1. 如果在 Python 程序中没有导入相关的模块(例如 import random、import math)，解释器将在运行时抛出________错误。

A. 语法　　B. 运行时　　C. 逻辑　　D. 不报错

2. 如果在 Python 程序中包括零除运算，解释器将在运行时抛出________错误信息。

A. NameError　　B. FileNotFoundError

C. SyntaxError　　D. ZeroDivisionError

3. 如果在 Python 程序中试图打开不存在的文件,解释器将在运行时抛出________错误信息。

A. NameError　　B. FileNotFoundError
C. SyntaxError　　D. ZeroDivisionError

4. 在 Python 程序中对于表达式 123 + 'xyz',解释器将抛出________错误信息。

A. NameError　　B. FileNotFoundError
C. SyntaxError　　D. TypeError

5. 在 Python 程序中假设列表 s=[1,2,3],如果语句中使用 s[3],则解释器将抛出________错误信息。

A. NameError　　B. IndexError　　C. SyntaxError　　D. TypeError

6. 在 Python 程序中假设字典 d={'1': 'male', '2': 'female'},如果语句中使用 d[3],则解释器将抛出________错误信息。

A. NameError　　B. IndexError　　C. KeyError　　D. TypeError

7. 以下关于异常处理 try 语句块的说法,不正确的是________。

A. finally 语句中的代码段始终要保证被执行
B. 一个 try 块后接一个或多个 except 块
C. 一个 try 块后接一个或多个 finally 块
D. try 块必须与 except 或 finally 块一起使用

二、填空题

1. Python 语言采用结构化的异常处理机制。在程序运行过程中如果产生错误,则抛出异常;通过________语句来定义代码块,以运行可能抛出异常的代码;通过________语句可以捕获特定的异常并执行相应的处理;通过________语句可以保证即使产生异常(处理失败),也可以在事后清理资源等。

2. 在某种特殊条件下,Python 代码中也可以创建一个异常对象,并通过________语句抛出给系统运行时。

3. 使用关键字________可以声明断言。

4. Python 解释器有调试模式和优化模式两种运行模式,当使用选项________运行时为优化模式,此时内置只读变量__debug__为________。

5. 自定义异常类一般继承于________或其子类。

三、思考题

1. Python 程序的错误通常可以分为哪 3 种类型?请分别举例说明。

2. 简述 Python 中 try…except…finally 各语句的用法和作用。

3. try 语句一般有哪几种可能的形式?

4. 断言的主要功能是什么?断言一般用于哪些情况?

5. Python 解释器有哪两种运行模式?如何设置 Python 解释器的运行模式?

7.7 上机实践

完成本章中的例 7.1~例 7.20,熟悉 Python 语言异常处理机制的捕获和处理。

7.8　案例研究：使用调试器调试 Python 程序

分析定位程序错误是程序设计最基本的功能，Python 标准库的调试器 pdb 提供了基本的调试功能，例如设置断点、查看变量等。

集成开发环境（IDE，例如 IDLE、Spyder、PyCharm）提供了更直接、方便的调试器。

本章案例研究通过实例阐述使用 IDLE 调试器跟踪调试 Python 程序的基本方法。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 8 章

函数和函数式编程


视频讲解

函数是可重用的程序代码段。在 Python 中有常用的内置函数，例如 len()、sum()等。在 Python 模块和程序中也可以自定义函数。使用函数可以提高编程效率。

8.1 函数概述

8.1.1 函数的基本概念

函数用于在程序中分离不同的任务。在程序设计过程中，如果可以分离任务，则建议使用函数分别实现分离后的子任务。

函数为代码复用提供了一个通用的机制，定义和使用函数是 Python 程序设计的重要组成部分。

函数允许程序的控制在调用代码和函数代码之间切换，也可以把控制转换到自身的函数，即函数自己调用本身，此过程称为递归(recursion)调用。

8.1.2 函数的功能

函数是模块化程序设计的基本构成单位，使用函数具有如下优点。

(1) 实现结构化程序设计：通过把程序分割为不同的功能模块可以实现自顶向下的结构化设计。

(2) 减少程序的复杂度：简化程序的结构，提高程序的可阅读性。

(3) 实现代码的复用：一次定义多次调用，实现代码的可重用性。

(4) 提高代码的质量：实现分割后子任务的代码相对简单，易于开发、调试、修改和维护。

(5) 协作开发：在将大型项目分割成不同的子任务后，团队多人可以分工合作，同时进行协作开发。

(6) 实现特殊功能：递归函数可以实现许多复杂的算法。

8.1.3 Python 中函数的分类

在 Python 语言中函数可以分为以下 4 类。

(1) 内置函数：Python 语言内置了若干常用的函数，例如 abs()、len()等，在程序中可以直接使用。

(2) 标准库函数：Python 语言安装程序同时会安装若干标准库，例如 math、random 等。

通过 import 语句可以导入标准库，然后使用其中定义的函数。

(3) 第三方库函数：Python 社区提供了许多其他高质量的库，例如 Python 图像库等。在下载、安装这些库后，通过 import 语句可以导入库，然后使用其中定义的函数。

(4) 用户自定义函数：本章将详细讨论函数的定义和调用方法。

8.2　函数的声明和调用

8.2.1　函数对象的创建

在 Python 语言中函数也是对象，使用 def 语句创建，其语法格式如下。

```
def 函数名([形参列表]):
    函数体
```

说明：

(1) 函数使用关键字 def 声明，函数名为有效的标识符(命名规则为全小写字母，可以使用下画线增加可阅读性，例如 my_func)，形参列表(用圆括号括起来，并用逗号隔开，可能为空)为函数的参数。函数定义的第一行称为函数签名(signature)，函数签名指定函数名称以及函数的每个形式参数变量名称。

(2) 在声明函数时可以声明函数的参数，即形式参数，简称形参；形参在函数定义的圆括号对内指定，用逗号分隔。在调用函数时需要提供函数所需参数的值，即实际参数，简称实参。

(3) def 是复合语句，故函数体需采用缩进书写规则。

(4) 函数可以使用 return 返回值。如果函数体中包含 return 语句，则返回值；否则不返回，即返回值为空(None)。无返回值的函数相当于其他编程语言中的过程。

(5) def 是执行语句，Python 解释执行 def 语句时会创建一个函数对象，并绑定到函数名变量。

【例 8.1】　函数创建示例 1：定义返回两个数的平均值的函数。

```
def my_average(a, b):
    return(a+b)/2
```

【例 8.2】　函数创建示例 2：定义打印 n 个星号的无返回值的函数。

```
def print_star(n):
    print(("*"*n).center(50))       #打印 n 个星号，两边填充空格，总宽度为 50
```

【例 8.3】　函数创建示例 3：定义计算并返回第 n 阶调和数(1＋1/2＋1/3＋…＋1/n)的函数。

```
def harmonic(n): #计算 n 阶调和数(1 + 1/2 + 1/3 + … + 1/n)
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total
```

8.2.2　函数的调用

在进行函数调用时，根据需要可以指定实际传入的参数值。函数调用的语法格式如下。

```
函数名([实参列表]);
```

说明：

（1）函数名是当前作用域中可用的函数对象，即调用函数之前程序必须先执行 def 语句，创建函数对象（内置函数对象会自动创建，import 导入模块时会执行模块中的 def 语句，创建模块中定义的函数）。函数的定义位置必须位于调用该函数的全局代码之前，故典型的 Python 程序结构顺序通常为①import 语句>②函数定义>③全局代码。

（2）实参列表必须与函数定义的形参列表一一对应，有关函数参数的详细信息请参见本章第 8.3 节内容。

（3）函数调用是表达式。如果函数有返回值，可以在表达式中直接使用；如果函数没有返回值，则可以单独作为表达式语句使用。

【例 8.4】 函数的调用示例 1（triangle.py）：先定义一个打印 n 个星号的无返回值的函数 print_star(n)，然后从命令行第一个参数中获取所需打印的三角形的行数 lines，并循环调用 print_star() 函数输出由星号构成的等腰三角形，每行打印 1，3，5，…，2 * lines−1 个星号。

```
import sys
def print_star(n):
    print(("*" * n).center(50))          #打印 n 个星号,两边填充空格,总宽度为 50
lines = int(sys.argv[1])                 #三角形的行数
for i in range(1, 2 * lines,2):          #每行打印 1、3、5、…、2 * lines - 1 个星号
    print_star(i)
```

程序运行结果如图 8-1 所示。

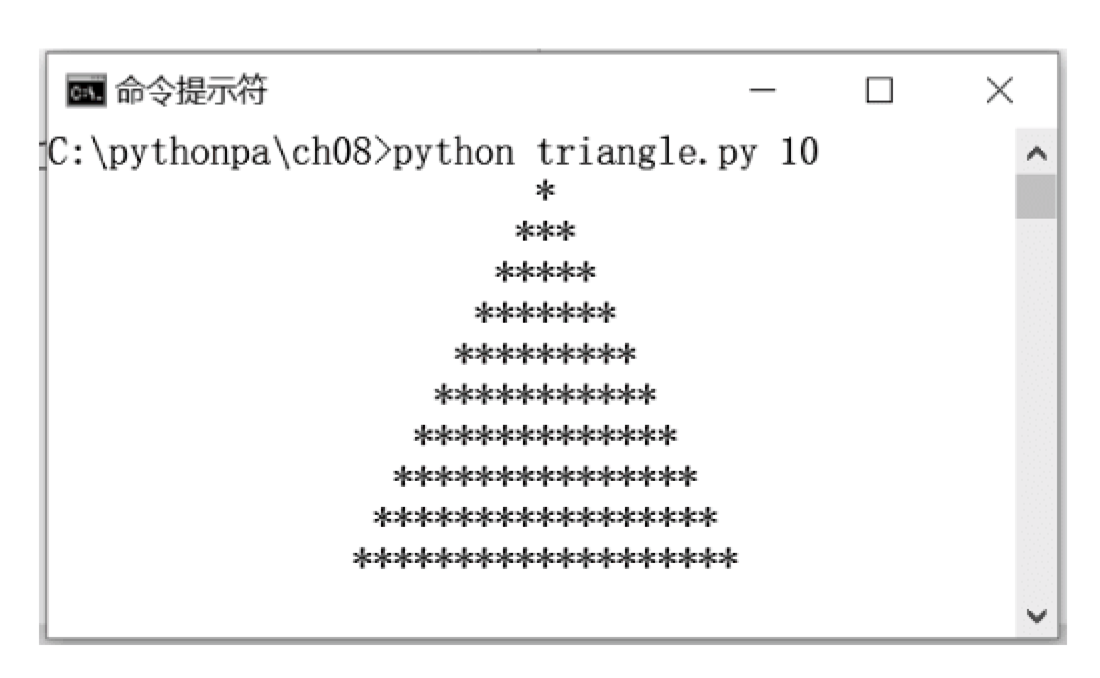


图 8-1 输出星号构成的三角形

【例 8.5】 函数的调用示例 2（harmonic.py）：定义计算并返回第 n 阶调和数（1 + 1/2 + 1/3 + … + 1/n）的函数，输出前 n 个调和数。

```
import sys
def harmonic(n):                         #计算 n 阶调和数(1 + 1/2 + 1/3 + … + 1/n)
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total
n = int(sys.argv[1])                     #从命令行第一个参数中获取调和数阶数
for i in range(1, n+1):                  #输出前 n 个调和数的值
    print(harmonic(i))
```

程序运行结果如图 8-2 所示。

8.2.3 函数的副作用

大多数函数接收一个或多个参数，通过计算返回一个值，这种类型的函数称为纯函数（pure function），即给定同样的实际参数，其返回值唯一，且不会产生其他的可观察到的副作

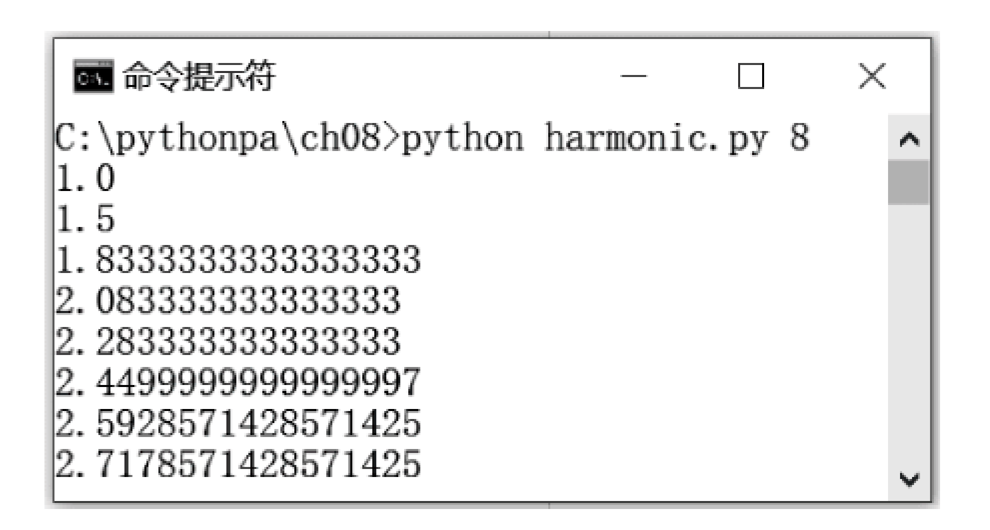


图 8-2　输出前 n 个调和数

用,例如读取键盘输入、产生输出、改变系统的状态等。

相对于纯函数,产生副作用的函数也有一定的应用。在一般情况下,产生副作用的函数相当于其他程序设计语言中的过程。在这些函数中可以省略 return 语句:当 Python 执行完函数的最后一条语句后,将控制权返回给调用者。

例如,函数 print_star(n)的副作用是向标准输出写入若干星号。

编写同时产生副作用和返回值的函数通常被认为是不良编程风格,但有一个例外,即读取函数。例如,input()函数既返回一个值,同时又产生副作用(从标准输入中读取并消耗一个字符串)。

8.3　参数的传递

8.3.1　形式参数和实际参数

函数的声明可以包含一个[形参列表],而函数调用则通过传递[实参列表],以允许函数体中的代码引用这些参数变量。

声明函数时所声明的参数即为形式参数,简称形参;调用函数时提供函数所需要的参数的值即为实际参数,简称实参。

实际参数值默认按位置顺序依次传递给形式参数。如果参数个数不对,将会产生错误。

【例 8.6】 形式参数和实际参数示例(my_max1.py)。

```
def my_max1(a, b):
    if a > b: print(a, '>', b)
    elif a == b: print(a, '=', b)
    else: print(a, '<', b)
my_max1(1, 2)
x = 11; y = 8
my_max1(x, y)
my_max1(1)
```

程序运行结果如下。

```
1 < 2
11 > 8
Traceback (most recent call last):
  File "C:\pythonpa\ch08\my_max1.py", line 8, in <module>
    my_max1(1)
TypeError: my_max1() missing 1 required positional argument: 'b'
```

8.3.2 形式参数变量和对象引用传递

声明函数时声明的形式参数等同于函数体中的局部变量，在函数体中的任何位置都可以使用。

局部变量和形式参数变量的区别在于局部变量在函数体中绑定到某个对象，而形式参数变量则绑定到函数调用代码传递的对应实际参数对象。

Python 参数传递方法是传递对象引用，而不是传递对象的值。

8.3.3 传递不可变对象的引用

在调用函数时，若传递的是不可变对象（例如 int、float、str 和 bool 对象）的引用，则如果函数体中修改对象的值，其结果实际上是创建了一个新的对象。

【例 8.7】 传递不可变对象的引用示例（inc1. py）：错误的递增函数。

```
i = 100
def inc(j,n):
    j += n
inc(i,10)
print(i)
```

在本例中，i 的初值为 100，当调用函数 inc(i,10)后，在函数体内执行了“i += 10”语句，函数体内的 i 变成了 110。但是，当函数调用完毕返回主程序时 i 的值仍然为 100，因为整数 i 是不可变对象，而在 Python 语言中一个函数不能改变一个不可变对象（例如整数、浮点数、布尔值或字符串）的值（即函数无法产生副作用）。

通过例 8.8 可以实现增量函数的功能。

【例 8.8】 传递不可变对象的引用示例（inc2. py）：正确的递增函数。

```
i = 100
def inc(j,n):
    j += n
    return j
i = inc(i,10)
print(i)
```

在本例中，i 的初值为 100，当使用表达式“i=inc(i,10)”调用函数 inc(i,10)后，在函数体内执行了“i += 10”语句，函数体内的 i 变成了 110，并且函数返回了 110。当函数调用完毕返回主程序时 i 被赋值为 110。

8.3.4 传递可变对象的引用

在调用函数时，如果传递的是可变对象（例如 list 对象）的引用，则在函数体中可以直接修改对象的值。

【例 8.9】 传递可变对象引用的函数示例 1：定义一个可以交换给定列表中两个指定下标的元素值的函数。

```
def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

【例 8.10】 传递可变对象引用的函数示例 2：随机混排给定列表的元素值。

```
def shuffle(a):
    n = len(a)                      # 获取列表 a 的长度 n
    for i in range(n):              # 从 0 到 n-1 进行循环迭代
        r = random.randrange(i, n)  # 取[i,n)的随机整数
        exchange(a, i, r)           # 交换列表 a 中下标分别为 i 和 r 的元素的值
```

8.3.5 可选参数

在声明函数时,如果希望函数的一些参数是可选的,可以在声明函数时为这些参数指定默认值。在调用该函数时,如果没有传入对应的实参值,则函数使用声明时指定的默认参数值。例如:

```
>>> def babble(words, times = 1):       # 声明函数 babble(),第二个参数指定了默认值
    print(words * times)
>>> babble('Hello')                     # 调用函数 babble(),只指定了一个参数传给 words,times 使
                                        # 用默认值 1
Hello
>>> babble('Tiger ', 3)                 # 调用函数 babble(),传'Tiger'给 words,传 3 给 times
Tiger Tiger Tiger
```

注意: 必须先声明没有默认值的形参,然后再声明有默认值的形参。这是因为在函数调用时默认是按位置传递实际参数值的。例如:

```
>>> def my_func(a, b=5):
        pass
>>> def my_func1(a=5, b):               # 默认值的形参位置不正确
        pass
SyntaxError: non-default argument follows default argument
```

【例 8.11】 可选参数示例(my_sum1.py):基于期中成绩和期末成绩,按照指定的权重计算总评成绩。

```
def my_sum1(mid_score, end_score, mid_rate = 0.4): #期中成绩、期末成绩、期中成绩权重
    #基于期中成绩、期末成绩和权重计算总评成绩
    score = mid_score * mid_rate + end_score * (1 - mid_rate)
    print(format(score, '.2f')) #输出总评成绩,保留两位小数
my_sum1(88, 79)                         #期中成绩权重为默认的 40%
my_sum1(88, 79, 0.5)                    #将期中成绩权重设置为 50%
```

程序运行结果如下。

```
82.60
83.50
```

8.3.6 位置参数和命名参数

在函数调用时,实参默认按位置顺序传递形参。按位置传递的参数称为位置参数。

在函数调用时,也可以通过名称(关键字)指定传入的参数,例如 my_max1(a=1, b=2)或者 my_max1(b=2, a=1)。

按名称指定传入的参数称为命名参数,也称为关键字参数。使用关键字参数具有 3 个优点:参数按名称意义明确;传递的参数与顺序无关;如果有多个可选参数,则可以选择指定某个参数值。

在带星号的参数后面声明的参数强制为命名参数,如果这些参数没有默认值,且调用时必

须使用命名参数赋值，则会引发错误。

如果不需要带星号的参数，只需要强制命名参数，则可以简单地使用一个星号，例如 def total(initial=5， * ，vegetables)。

【例 8.12】 命名参数示例(my_sum2.py)：基于期中成绩和期末成绩，按照指定的权重计算总评成绩。本例中所使用的 3 种调用方式等价。

```
def my_sum2(mid_score, end_score, mid_rate = 0.4): #期中成绩、期末成绩、期中成绩权重
    #基于期中成绩、期末成绩和权重计算总评成绩
    score = mid_score * mid_rate + end_score * (1 - mid_rate)
    print(format(score, '.2f')) #输出总评成绩，保留两位小数
#期中 88，期末 79，并且期中成绩权重为默认的 40%.3 种调用方式等价
my_sum2(88, 79)
my_sum2(mid_score = 88, end_score = 79)
my_sum2(end_score = 79, mid_score = 88)
```

程序运行结果如下。

```
82.60
82.60
82.60
```

8.3.7 可变参数

在声明函数时，可以通过带星的参数(例如 * param1)向函数传递可变数量的实参。在调用函数时，从那一点后所有的参数被收集为一个元组。

在声明函数时，也可以通过带双星的参数(例如 ** param2)向函数传递可变数量的实参。在调用函数时，从那一点后所有的参数被收集为一个字典。

带星或带双星的参数必须位于形参列表的最后位置。

【例 8.13】 可变参数示例 1(my_sumVarArgs1.py)：利用带星的参数计算各数字的累加和。

```
def my_sum3(a, b, *c):                        #各数字的累加和
    total = a + b
    for n in c:
        total = total + n
    return total
print(my_sum3(1, 2))                          #计算 1+2
print(my_sum3(1, 2, 3, 4, 5))                 #计算 1+2+3+4+5
print(my_sum3(1, 2, 3, 4, 5, 6, 7))           #计算 1+2+3+4+5+6+7
```

程序运行结果如下。

```
3
15
28
```

【例 8.14】 可变参数示例 2(my_sumVarArgs2.py)：利用带星和带双星的参数计算各数字的累加和。

```
def my_sum4(a, b, *c, **d):                   #各数字的累加和
    total = a + b
    for n in c:                               #元组中各元素的累加和
        total = total + n
    for key in d:                             #字典中各元素的累加和
```

```
        total = total + d[key]
    return total
print(my_sum4(1, 2))                                #计算 1+2
print(my_sum4(1, 2, 3, 4, 5))                       #计算 1+2+3+4+5
print(my_sum4(1, 2, 3, 4, 5, male = 6, female = 7)) #计算 1+2+3+4+5+6+7
```

程序运行结果如下。

```
3
15
28
```

8.3.8　强制命名参数

在带星号的参数后面声明参数会导致强制命名参数(Keyword-only)。在调用时必须显式使用命名参数传递值,因为按位置传递的参数默认收集为一个元组,传递给前面带星号的可变参数。

如果不需要带星号的可变参数,只想使用强制命名参数,可以简单地使用一个星号,例如 def my_func(* , a, b, c)。

【例 8.15】 强制命名参数示例(keyword_only.py):基于期中成绩和期末成绩,按照指定的权重计算总评成绩。

```
def my_sum( * , mid_score, end_score, mid_rate = 0.4):#期中成绩、期末成绩、期中成绩权重
    #基于期中成绩、期末成绩和权重计算总评成绩
    score = mid_score * mid_rate + end_score * (1 - mid_rate)
    print(format(score, '.2f'))                      #输出总评成绩,保留两位小数
my_sum(mid_score = 88, end_score = 79)               #期中 88,期末 79,期中权重为默认的 40%
my_sum(end_score = 79, mid_score = 88)               #期末 79,期中 88,期中权重为默认的 40%
my_sum(88, 79)                                       #报错,必须显式使用命名参数传递值
```

程序运行结果如下。

```
82.60
82.60
Traceback(most recent call last):
  File "C:\pythonpa\ch08\keyword_only.py", line 7, in <module>
    my_sum(88, 79)                                   #报错,必须显式使用命名参数传递值
TypeError: my_sum() takes 0 positional arguments but 2 were given
```

8.3.9　参数类型检查

通常,函数在定义时既要指定定义域也要指定值域,即指定形式参数和返回值的类型。

基于 Python 语言的设计理念,在定义函数时不用限定其参数和返回值的类型。这种灵活性可以实现多态性,即允许函数适用于不同类型的对象,例如 my_average(a,b)函数,既可以返回两个 int 对象的平均值,也可以返回两个 float 对象的平均值。

当使用不支持的类型参数调用函数时会产生错误。例如,my_average(a,b)函数传递的参数为 str 对象,Python 在运行时将抛出错误 TypeError。

原则上可以增加代码检测这种类型错误,但 Python 程序设计遵循一种惯例,即用户调用函数时必须理解并保证传入正确类型的参数值。本书实现的函数均采用这种设计理念。

8.4 函数的返回值

8.4.1 return 语句和函数返回值

在函数体中使用 return 语句可以实现从函数中返回一个值并跳出函数的功能。

【例 8.16】 函数的返回值示例(my_max.py)：编写函数，利用 return 语句返回函数值，求若干数中的最大值。

求若干数中最大值的方法一般如下。

(1) 将最大值的初值设为一个比较小的数，或者取第一个数为最大值的初值。

(2) 利用循环将每个数与最大值比较，若此数大于最大值，则将此数设置为最大值。

```
def my_max(a, b, *c):              #求若干数中的最大值
    max_value = a                  #假设第一个数为最大值
    if max_value < b:              #如果最大值小于 b,则 b 为最大值
        max_value = b
    for n in c:                    #循环迭代 c 中的每个元素 n,如果最大值小于 n,则 n 为最大值
        if max_value < n:
            max_value = n
    return max_value               #利用 return 语句返回最大值
#测试代码
print(my_max(1, 2))                #求(1, 2)中的最大值
print(my_max(1, 7, 11, 2, 5))      #求(1, 7, 11, 2, 5)中的最大值
```

程序运行结果如下。

```
2
11
```

8.4.2 多条 return 语句

return 语句可以放置在函数中的任何位置，当执行到第一个 return 语句时程序返回到调用程序。

【例 8.17】 判断素数示例(prime.py)：先编写判断一个数是否为素数的函数，然后编写测试代码，判断并输出 1～99 中的素数。

所谓素数（或称质数），是指除了 1 和该数本身，不能被任何整数整除的正整数。判断一个正整数 n 是否为素数，只要判断 n 可否被 2～$\sqrt{n}$中的任何一个整数整除，如果 n 不能被此范围中的任何一个整数整除，n 即为素数，否则 n 为合数。

```
def is_prime(n):
    if n < 2: return False         #如果 n 小于 2,返回 False
    i = 2
    while i * i <= n:
        #一旦 n 能够被 2～√n中的任意整数整除,n 就不是素数,返回 False
        if n % i == 0: return False
        i += 1
    return True
#测试代码
for i in range(100):               #判断并输出 1～99 中的素数,以空格分隔
    if is_prime(i):print(i, end = ' ')
```

程序运行结果如下。

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

8.4.3 返回多个值

在函数体中使用 return 语句可实现从函数返回一个值并跳出函数。如果需要返回多个值,则可以返回一个元组。

【例 8.18】 编写函数,返回一个随机列表(randomarray.py)。先编写一个函数,生成由 n 个随机整数构成的列表,然后编写测试代码,生成并输出由 5 个随机整数构成的列表的各元素值。

```
import random
def randomarray(n):                 #生成由 n 个随机数构成的列表
    a = []
    for i in range(n):
        a.append(random.random())
    return a
#测试代码
b = randomarray(5)                  #生成由 5 个随机数构成的列表
for i in b: print(i)                #输出列表中的每个元素
```

程序运行结果如下(每次运行结果为随机数)。

```
0.307835337127647
0.0869723095733228
0.648192164694294
0.26651844944908465
0.12234774081646149
```

8.5 变量的作用域

变量声明的位置不同,其可以被访问的范围也不同。变量的可被访问范围称为变量的作用域。变量按其作用域大致可以分为全局变量、局部变量和类成员变量。

8.5.1 全局变量

在一个源代码文件中,在函数和类定义之外声明的变量称为全局变量。全局变量的作用域为其定义的模块,从定义的位置起,直到文件结束位置。

通过 import 语句导入模块,也可以通过全限定名称"**模块名.变量名**"访问;或者通过 from…import 语句导入模块中的变量并访问。

不同的模块都可以访问全局变量,这会导致全局变量的不可预知性。如果多个语句同时修改一个全局变量,则可能导致程序产生错误,且很难发现和更正。

全局变量降低了函数或模块之间的通用性,也降低了代码的可读性。在一般情况下,应该尽量避免使用全局变量。全局变量一般作为常量使用。

【例 8.19】 全局变量定义示例(global_variable.py)。

```
TAX1 = 0.17                         #税率常量 17%
TAX2 = 0.2                          #税率常量 20%
TAX3 = 0.05                         #税率常量 5%
PI = 3.14                           #圆周率 3.14
```

【例 8.20】 全局变量使用示例(tax.py)。

```
import global_variable                  #导入全局变量定义
def tax(x):                             #根据税率常量 20% 计算纳税值
     return x * global_variable.TAX2
#测试代码
a = [1000, 1200, 1500, 2000]
for i in a:                             #计算并打印 4 笔数据的纳税值
     print(i, tax(i))
```

程序运行结果如下。

```
1000 200.0
1200 240.0
1500 300.0
2000 400.0
```

8.5.2 局部变量

在函数体中声明的变量(包括函数参数)称为局部变量,其有效范围(作用域)为函数体。

全局代码不能引用一个函数的局部变量或形式参数变量;一个函数也不能引用在另一个函数中定义的局部变量或形式参数变量。

如果在一个函数中定义的局部变量(或形式参数变量)与全局变量重名,则局部变量(或形式参数变量)优先,即函数中定义的变量是指局部变量(或形式参数变量),而不是全局变量。

【例 8.21】 局部变量定义示例(local_variable.py)。

```
num = 100                               #全局变量
def f():
    num = 105                           #局部变量
    print(num)                          #输出局部变量的值
#测试代码
f();print(num)
```

程序运行结果如下。

```
105
100
```

说明: 函数 f()中的 print(num)语句引用的是局部变量 num,因此输出 105。

8.5.3 全局语句 global

在函数体中可以引用全局变量,但如果函数内部的变量名是第一次出现且在赋值语句之前(变量赋值),则解释为定义局部变量。

【例 8.22】 函数体错误引用全局变量的示例(f_global.py)。

```
m = 100
n = 200
def f():
     print(m+5)                         #引用全局变量 m
     n += 10                            #错误,n 在赋值语句前面,解释为局部变量(不存在)
#测试代码
f()
```

程序运行结果如下。

```
105
Traceback(most recent call last):
  File "C:\pythonpa\ch08\f_global.py", line 7, in <module>
    f()
  File "C:\pythonpa\ch08\f_global.py", line 5, in f
    n += 10                          #错误,n在赋值语句前面,解释为局部变量(不存在)
UnboundLocalError: local variable 'n' referenced before assignment
```

如果要为定义在函数外的全局变量赋值,可以使用 global 语句,表明变量是在外面定义的全局变量。global 语句可以指定多个全局变量,例如“global x, y, z”。一般应该尽量避免这样使用全局变量,全局变量会导致程序的可读性差。

【例 8.23】 全局语句 global 示例(global.py)。

```
pi = 3.141592653589793              #全局变量
e = 2.718281828459045               #全局变量
def my_func():
     global pi                      #全局变量,与前面的全局变量 pi 指向相同的对象
     pi = 3.14                      #改变了全局变量的值
     print('global pi = ', pi)      #输出全局变量的值
     e = 2.718                      #局部变量,与前面的全局变量 e 指向不同的对象
     print('local e = ', e)         #输出局部变量的值
#测试代码
print('module pi = ', pi)           #输出全局变量的值
print('module e = ', e)             #输出全局变量的值
my_func()                           #调用函数
print('module pi = ', pi)           #输出全局变量的值,该值在函数中已被更改
print('module e = ', e)             #输出全局变量的值
```

程序运行结果如下。

```
module pi = 3.141592653589793
module e = 2.718281828459045
global pi = 3.14
local e = 2.718
module pi = 3.14
module e = 2.718281828459045
```

8.5.4 非局部语句 nonlocal

在函数体中可以定义嵌套函数,在嵌套函数中如果要为定义在上级函数体的局部变量赋值,可以使用 nonlocal 语句,表明变量不是所在块的局部变量,而是在上级函数体中定义的局部变量。nonlocal 语句可以指定多个非局部变量,例如“nonlocal x, y, z”。

【例 8.24】 非局部语句 nonlocal 示例(nonlocal.py)。

```
def outer_func():
     tax_rate = 0.17                           #上级函数体中的局部变量
     print('outer func tax rate = ', tax_rate) #输出上级函数体中局部变量的值
     def innner_func():
         nonlocal tax_rate                     #不是所在块的局部变量,而是在上级函数体
                                               #中定义的局部变量
         tax_rate = 0.05                       #上级函数体中的局部变量重新赋值
         print('inner func tax rate = ', tax_rate) #输出上级函数体中局部变量的值
     innner_func()                             #调用函数
     print('outer func tax rate = ', tax_rate) #输出上级函数体中局部变量的值(已更改)
#测试代码
```

```
outer_func()
```

程序运行结果如下。

```
outer func tax rate = 0.17
inner func tax rate = 0.05
outer func tax rate = 0.05
```

8.5.5 类成员变量

类成员变量是在类中声明的变量，包括静态变量和实例变量，其有效范围(作用域)为类定义体内。

在外部，通过创建类的对象实例，然后通过“对象.实例变量”访问类的实例变量，或者通过“类.静态变量”访问类的静态变量。

具体请参见第 9 章。

8.5.6 输出局部变量和全局变量

在程序运行过程中，在上下文中会生成各种局部变量和全局变量，使用内置函数 globals()和 locals()可以查看并输出局部变量和全局变量列表。

【例 8.25】 局部变量和全局变量列表示例(locals_globals.py)。

```
a = 1
b = 2
def f(a, b):
    x = 'abc'
    y = 'xyz'
    for i in range(2): # i = 0～1
        j = i
        k = i * * 2
        print(locals())
#测试代码
f(1,2)
print(globals())
```

程序运行结果如下。

```
{'a': 1, 'b': 2, 'x': 'abc', 'y': 'xyz', 'i': 0, 'j': 0, 'k': 0}
{'a': 1, 'b': 2, 'x': 'abc', 'y': 'xyz', 'i': 1, 'j': 1, 'k': 1}
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen
_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module '
builtins' (built-in)>, '__file__': 'C:\\pythonpa\\ch08\\locals_globals.py', 'a': 1,'b': 2,
'f ':< function f at 0x000002274714BE18 >}
```

8.6 递 归 函 数

8.6.1 递归函数的定义

递归函数即自调用函数，在函数体内部直接或间接地自己调用自己，即函数的嵌套调用是函数本身。递归函数常用来实现数值计算的方法。

例如，非负整数的阶乘定义为：

$n! = n\times(n-1)\times(n-2)\times\cdots\times2\times1$，当 n=1 时，n! = 1

即 n! 是所有小于或等于 n 的正整数的乘积。很显然,使用 for 循环结构能很容易地计算 n!。更简单的方法是采用递归函数实现:

```
n!  =  1                 #当 n == 1 时
n!  =  n×(n-1)!          #当 n>1 时
```

【例 8.26】 使用递归函数实现阶乘(factorial.py)。

```
def factorial(n):
    if n == 1: return 1
    return n * factorial(n - 1)
#测试代码
for i in range(1,10):     #输出 1~9 的阶乘
    print(i,'! =', factorial(i))
```

程序运行结果如下。

```
1 ! = 1
2 ! = 2
3 ! = 6
4 ! = 24
5 ! = 120
6 ! = 720
7 ! = 5040
8 ! = 40320
9 ! = 362880
```

8.6.2 递归函数的原理

递归提供了建立数学模型的一种直接方法,与数学上的数学归纳法相对应。

每个递归函数必须包括以下两个主要部分。

(1) 终止条件:表示递归的结束条件,用于返回函数值,不再递归调用。例如,factorial()函数的结束条件为“n 等于 1”。

(2) 递归步骤:递归步骤把第 n 步的参数值的函数与第 n-1 步的参数值的函数关联。例如,对于 factorial(),其递归步骤为“n * factorial(n-1)”。

另外,一序列的参数值必须逐渐收敛到结束条件。例如,对于 factorial(),每次递归调用参数值 n 均递减 1,所以一序列参数值逐渐收敛到结束条件(n=1)。

例如,调和数的计算公式如下。

$$H_n = 1 + 1/2 + \cdots + 1/n$$

故可以使用递归函数实现。

(1) 终止条件:$H_n = 1$　　　　当 n==1 时

(2) 递归步骤:$H_n = H_{n-1} + 1/n$　　当 n>1 时

每次递归,n 严格递减,故逐渐收敛于 1。

【例 8.27】 使用递归函数实现调和数(harmonicRecursion.py)。

```
def harmonic(n):
    if n == 1: return 1.0                #终止条件
    return harmonic(n-1) + 1.0/n         #递归步骤
#测试代码
for i in range(1,10):                    #输出 1~9 阶的调和数
    print('H', i, ' = ', harmonic(i))
```

程序运行结果如下。

```
H 1 = 1.0
H 2 = 1.5
H 3 = 1.8333333333333333
H 4 = 2.083333333333333
H 5 = 2.283333333333333
H 6 = 2.4499999999999997
H 7 = 2.5928571428571425
H 8 = 2.7178571428571425
H 9 = 2.8289682539682537
```

8.6.3 编写递归函数时需要注意的问题

虽然使用递归函数可以实现简洁、优雅的程序,但在编写递归函数时应该注意如下几个问题。

(1) 必须设置终止条件。

缺少终止条件的递归函数将导致无限递归函数调用,其最终结果是系统会耗尽内存。此时 Python 会抛出错误 RuntimeError,并报告错误信息"maximum recursion depth exceeded(超过最大递归深度)"。

在递归函数中一般需要设置终止条件。在 sys 模块中,函数 getrecursionlimit()和 setrecursionlimit()用于获取和设置最大递归次数。例如:

```
>>> import sys
>>> sys.getrecursionlimit()          #获取最大递归次数:1000
>>> sys.setrecursionlimit(2000)      #设置最大递归次数为 2000
```

(2) 必须保证收敛。

递归调用所解决子问题的规模必须小于原始问题的规模,否则会导致无限递归函数调用。

(3) 必须保证内存和运算消耗控制在一定范围内。

递归函数代码虽然看起来简单,但往往会导致过量的递归函数调用,从而消耗过量的内存(导致内存溢出),或过量的运算能力(运行时间过长)。

8.6.4 递归函数的应用:最大公约数

用于计算最大公约数问题的递归方法称为欧几里得算法,其描述如下:

如果 p>q,则 p 和 q 的最大公约数等于 q 和 p % q 的最大公约数。

故可以使用递归函数实现,步骤如下。

(1) 终止条件:gcd(p, q) = p　　#当 q==0 时

(2) 递归步骤:gcd(q, p%q)　　#当 q>1 时

每次递归,p%q 严格递减,故逐渐收敛于 0。

【例 8.28】 使用递归函数计算最大公约数(gcd.py)。

```
import sys
def gcd(p, q): #使用递归函数计算 p 和 q 的最大公约数
    if q == 0: return p                #如果 q = 0,返回 p
    return gcd(q, p % q)               #否则,递归调用 gcd(q, p % q)
#测试代码
p = int(sys.argv[1])                   #p = 命令行的第一个参数
q = int(sys.argv[2])                   #q = 命令行的第二个参数
```

```
print(gcd(p, q))                    #计算并输出 p 和 q 的最大公约数
```

程序运行结果如图 8-3 所示。

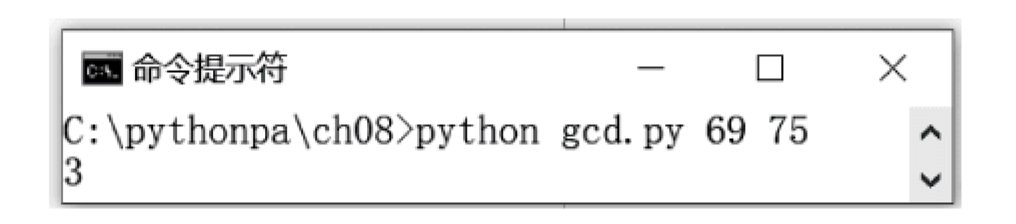


图 8-3　使用递归函数计算最大公约数

8.6.5　递归函数的应用：汉诺塔

汉诺塔(Towers of Hanoi，又称河内塔)源自于印度的古老传说：大梵天创造世界的时候，在世界中心贝拿勒斯的圣庙里做了 3 根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘，称之为汉诺塔。

大梵天命令婆罗门把圆盘从一根柱子上按大小顺序重新摆放到另一根柱子上，并且规定在 3 根柱子之间一次只能移动一个圆盘，且小圆盘上不能放置大圆盘。这个游戏称为汉诺塔益智游戏。

汉诺塔益智游戏问题很容易使用递归函数实现。假设柱子的编号为 a、b、c，定义函数 hanoi(n，a，b，c)表示把 n 个圆盘从柱子 a 移到柱子 c(可以经由柱子 b)，则有：

(1) 终止条件。当 n==1 时，hanoi(n，a，b，c)为终止条件。即如果柱子 a 上只有一个圆盘，则可以直接将其移动到柱子 c 上。

(2) 递归步骤。hanoi(n，a，b，c)可以分解为 3 个步骤，即 hanoi(n−1,a,c,b)、hanoi(1,a,b,c)和 hanoi(n−1,b,a,c)。如果柱子 a 上有 n 个圆盘，可以看成柱子 a 上有一个圆盘(底盘)和(n−1)个圆盘，首先需要把柱子 a 上面的(n−1)个圆盘移动到柱子 b，即调用 hanoi(n−1,a,c,b)；然后把柱子 a 上的最后一个圆盘移动到柱子 c，即调用 hanoi(1,a,b,c)；再将柱子 b 上的(n−1)个圆盘移动到柱子 c，即调用 hanoi(n−1,b,a,c)。

每次递归，n 严格递减，故逐渐收敛于 1。

【例 8.29】　使用递归函数实现汉诺塔问题(hanoi.py)。

```
#将 n 个从小到大依次排列的圆盘从柱子 a 移动到柱子 c 上，柱子 b 作为中间缓冲
def hanoi(n,a,b,c):
    if n == 1: print(a,'->',c)        #只有一个圆盘，直接将圆盘从柱子 a 移动到柱子 c 上
    else:
        hanoi(n-1,a,c,b)              #先将 n-1 个圆盘从柱子 a 移动到柱子 b 上(采用递归方式)
        hanoi(1,a,b,c)                #然后将最大的圆盘从柱子 a 移动到柱子 c 上
        hanoi(n-1,b,a,c)              #再将 n-1 个圆盘从柱子 b 移动到柱子 c 上(采用递归方式)
#测试代码
hanoi(4,'A','B','C')
```

程序运行结果如下。

```
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
A -> C
```

```
B -> C
B -> A
C -> A
B -> C
A -> B
A -> C
B -> C
```

8.7 内置函数的使用

在 Python 语言中提供了若干内置函数,用于实现常用的功能,可以直接使用。

8.7.1 内置函数一览

Python 中的内置函数如表 8-1 所示。

表 8-1 内置函数

内置函数				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	Reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

8.7.2 eval()函数

使用内置的 eval()函数可以对动态表达式进行求值,其语法形式如下。

```
eval(expression, globals = None, locals = None)
```

其中,expression 是动态表达式的字符串;globals 和 locals 是求值时使用的上下文环境的全局变量和局部变量,如果不指定,则使用当前运行上下文。例如:

```
>>> x = 2
>>> str_func = input("请输入表达式:")
请输入表达式:x * * 2 + 2 * x + 1
>>> eval(str_func)                          #对表达式 2 * * 2 + 2 * 2 + 1 求值.输出:9
```

eval()函数的功能是将字符串生成语句执行,如果字符串中包含不安全的语句(例如删除文件的语句),则存在注入安全隐患。

8.7.3　exec()函数

使用内置的 exec()函数可以执行动态语句，其语法形式如下。

```
exec(str[, globals[, locals]])
```

其中，str 是动态语句的字符串；globals 和 locals 是使用的上下文环境的全局变量和局部变量，如果不指定，则使用当前运行上下文。

通常，eval()用于动态表达式求值，返回一个值；exec()用于动态语句的执行，不返回值。同样，exec()也存在注入安全隐患。例如：

```
>>> exec("for i in range(10): print(i, end = ' ')")    #输出:0 1 2 3 4 5 6 7 8 9
```

8.7.4　compile()函数

使用内置的 compile()函数可以编译代码为代码对象，其语法形式如下。

```
compile(source, filename, mode)                        #返回代码对象
```

其中，source 为代码语句的字符串；如果是多行语句，则每一行的结尾必须有换行符\n。filename 为包含代码的文件。mode 为编译模式，可以为'exec'(用于语句序列的执行)、'eval'(用于表达式求值)和'single'(用于单个交互语句)。

编译后的代码对象可以通过 eval()函数或 exec()函数执行，因编译为代码对象，所以可以提高效率。例如：

```
>>> co = compile("for i in range(10): print(i, end = ' ')", '', 'exec')
>>> exec(co)          #输出:0 1 2 3 4 5 6 7 8 9
```

8.8　Python 函数式编程基础

Python 是面向对象的程序设计语言，也是面向过程的程序语言，同时也支持函数式编程。Python 标准库 functools 提供了若干关于函数的函数，提供了 Haskell 和 Standard ML 中的函数式程序设计工具。

8.8.1　作为对象的函数

在 Python 语言中函数也是对象，故函数对象可以赋值给变量。

【例 8.30】　作为对象的函数示例。

```
>>> f = abs
>>> type(f)                        #输出:<class 'builtin_function_or_method'>
>>> f(-123)                        #返回绝对值.输出:123
```

8.8.2　高阶函数

函数对象也可以作为参数传递给函数，还可以作为函数的返回值。参数为函数对象的函数或返回函数对象的函数称为高阶函数，即函数的函数。

【例 8.31】　高阶函数示例。

```
>>> def compute(f, s):             #f 为函数对象,s 为系列对象
        return f(s)
```

```
>>> compute(min, (1, 5, 3, 2))              #返回序列的最小值.输出:1
>>> compute(max, (1, 5, 3, 2))              #返回序列的最大值.输出:5
```

8.8.3 map()函数

在 Python 3 中,map()函数实现为内置的 map(f, iterable, …)可迭代对象(参见第 9 章),将函数 f 应用于可迭代对象,返回结果为可迭代对象。

【例 8.32】 map()函数示例 1:自定义函数 is_odd,应用该函数到可迭代对象的每一个元素,返回是否为奇数的可迭代对象结果。

```
>>> def is_odd(x):
        return x % 2 == 1
>>> list(map(is_odd, range(5)))             #输出:[False, True, False, True, False]
```

【例 8.33】 map()函数示例 2:使用内置函数 abs 返回绝对值列表。

```
>>> list(map(abs, [1, -3, 5, 6, -2, 4])) #输出:[1, 3, 5, 6, 2, 4]
```

【例 8.34】 map()函数示例 3:使用内置函数 str 返回元素的字符串表示形式。

```
>>> list(map(str, [1,2,3,4,5]))             #输出:['1', '2', '3', '4', '5']
```

【例 8.35】 map()函数示例 4:使用带两个参数的自定义函数,实现两个列表的元素依次比较的运算结果。

```
>>> def greater(x, y):
         return x > y
>>> list(map(greater,[1,5,7,3,9],[2,8,4,6,0]))
[False, False, True, False, True]
```

8.8.4 filter()函数

在 Python 3 中,filter()函数实现为内置的 filter(f, iterable)可迭代对象(参见第 9 章),将函数 f 应用于每个元素,然后根据返回值是 True 还是 False 决定保留还是丢弃该元素,返回结果为可迭代对象。

【例 8.36】 filter()函数示例 1:返回奇数的可迭代对象。

```
>>> def is_odd(x):
        return x % 2 == 1
>>> list(filter(is_odd, range(10)))         #输出:[1, 3, 5, 7, 9]
```

【例 8.37】 filter()函数示例 2:返回三位数的回文数(正序和反序相同)可迭代对象。

```
>>> def is_palindrome(x):
        if str(x) == str(x)[::-1]:
          return x
>>> list(filter(is_palindrome, range(100,1000)))
[101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 202, 212, 222, 232, 242, 252, 262, 272, 282,
292, 303, 313, 323, 333, 343, 353, 363, 373, 383, 393, 404, 414, 424, 434, 444, 454, 464, 474,
484, 494, 505, 515, 525, 535, 545, 555, 565, 575, 585, 595, 606, 616, 626, 636, 646, 656, 666,
676, 686, 696, 707, 717, 727, 737, 747, 757, 767, 777, 787, 797, 808, 818, 828, 838, 848, 858,
868, 878, 888, 898, 909, 919, 929, 939, 949, 959, 969, 979, 989, 999]
```

8.8.5 Lambda 表达式和匿名函数

在上述例子中,作为参数传递的函数可以为内置函数,也可以为自定义函数。如果函数的形式比较简单且只需要作为参数传递给其他函数,则可使用 Lambda 表达式直接定义匿名函

数。匿名函数广泛用于需要函数对象作为参数、函数比较简单并且只使用一次的场合。例如，filter(f, a)的第一个参数为函数对象，根据函数筛选列表的内容。

Lambda 是一种简便的、在同一行中定义函数的方法。Lambda 实际上生成一个函数对象，即匿名函数。

Lambda 表达式的基本格式如下。

```
lambda arg1,arg2… : <expression>
```

其中，arg1、arg2 等为函数的参数，<expression>为函数的语句，其结果为函数的返回值。例如，语句"lambda x,y: x + y"生成一个函数对象，函数参数为"x, y"，返回值为 x+y。

【例 8.38】 匿名函数示例 1。

```
>>> f = lambda x,y: x + y
>>> type(f)                                    #输出:<class 'function'>
>>> f(12, 34)                                  #计算两数之和.输出:46
```

【例 8.39】 匿名函数应用示例 1：过滤列表，返回元素为奇数的可迭代对象。

```
>>> list(filter(lambda x:x % 2 == 1, range(10)))  #输出:[1, 3, 5, 7, 9]
```

【例 8.40】 匿名函数应用示例 2：过滤列表，返回元素非空的可迭代对象。

```
>>> list(filter(lambda s:s and s.strip(), ['A', '', 'B', None, 'C', ' ']))
['A', 'B', 'C']
```

【例 8.41】 匿名函数应用示例 3：过滤列表，返回元素大于零的可迭代对象。

```
>>> list(filter(lambda x:x > 0, [1,0,-2,8,5]))  #输出:[1, 8, 5]
```

【例 8.42】 匿名函数应用示例 4：过滤列表，返回元素平方的可迭代对象。

```
>>> list(map(lambda x:x * x, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

8.8.6 operator 模块和操作符函数

operator 模块是 Python 内置操作符的函数接口，它定义了对应算术和比较等操作的函数，用于 map()、filter()等需要传递函数对象作为参数的场合，可以直接使用而不需要使用函数定义或者 Lambda 表达式，使得代码更加简洁。

【例 8.43】 查看 operator 模块中的函数对象。

```
>>> import operator
>>> dir(operator)
['__abs__', …, '_abs', 'abs', 'add', 'and_', 'attrgetter', 'concat', 'contains', 'countOf', 'delitem',
'eq', 'floordiv', 'ge', 'getitem', 'gt', 'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imatmul', 'imod', 'imul',
 'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter', '
itruediv', 'ixor', 'le', 'length_hint', 'lshift', 'lt', 'matmul', 'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not
_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
```

【例 8.44】 operator 模块应用示例 1：使用 operator 模块中的函数代替对应的运算符，concat(s1, s2)对应于 s1+s2。

```
>>> import operator
>>> a = 'hello'
>>> operator.concat(a, ' world')               #输出:'hello world'
```

【例 8.45】 operator 模块应用示例 2：在 map()中使用 operator.gt 函数对象(对应于操

作符>)实现两个列表的元素比较运算。

```
>>> import operator
>>> list(map(operator.gt,[1,5,7,3,9],[2,8,4,6,0]))
[False, False, True, False, True]
```

8.8.7 functools.reduce()函数

在 Python 3 中,reduce()函数实现为 functools.reduce(f, iterable[, initializer])函数,使用指定的带两个参数的函数 f 对一个数据集合(可迭代对象)的所有数据进行下列操作:使用第一个和第二个数据作为参数用 func()函数运算,得到的结果再与第三个数据作为参数用 func()函数运算,依此类推,最后得到一个结果。f 为函数对象;iterable 为可迭代对象;可选的 initializer 为初始值。

【例 8.46】 functools.reduce()示例 1:计算累加和。

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5])        # ((((1+2)+3)+4)+5) = 15
>>> functools.reduce(operator.add, [1, 2, 3, 4, 5], 10)    #10+ ((((1+2)+3)+4)+5) = 25
>>> functools.reduce(operator.add, range(1, 101))          #输出:5050
```

【例 8.47】 functools.reduce()示例 2:计算累乘结果。

```
>>> functools.reduce(operator.mul, range(1, 11))           #输出:3628800
```

8.8.8 偏函数

functools.partial()通过把一个函数的部分参数设置为默认值的方式返回一个新的可调用(callable)的 partial 对象,其语法形式如下。

```
functools.partial(func, *args, **keywords)
```

其中,func 为函数;args 为其位置参数;keywords 为关键字参数。

partial()函数主要用于设置预先已知的参数,从而减少调用时传递参数的个数。

【例 8.48】 functools.partial()应用示例:2 的 n 次方。

```
>>> import functools, math
>>> pow2 = functools.partial(math.pow, 2)          #封装 pow(x, y[, z]),指定参数 x=2
>>> list(map(pow2, range(11)))                     #输出 2 的 0~10 次方
[1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0, 1024.0]
```

8.8.9 sorted()函数

内置函数 sorted()把一个可迭代对象进行排序,返回结果列表。其语法形式如下。

```
sorted(iterable, *, key=None, reverse=False)
```

其中,iterable 是待排序的可迭代对象;key 是比较函数(默认为 None,按自然顺序排序);reverse 用于指定是否逆序排序。

【例 8.49】 sorted()函数示例。

```
>>> sorted([1,6,4,-2,9])                          #按数值自然排序:[-2, 1, 4, 6, 9]
>>> sorted([1,6,4,-2,9], reverse=True)            #按数值逆序排序:[9, 6, 4, 1, -2]
>>> sorted([1,6,4,-2,9], key=abs)                 #按绝对值排序:[1, -2, 4, 6, 9]
>>> sorted(["Dog","cat","Rabbit"])                #按字符串字典序排序:['Dog', 'Rabbit', 'cat']
>>> sorted(["Dog","cat","Rabbit"], key=str.lower) #字符串排序不区分大小写
```

```
['cat', 'Dog', 'Rabbit']
>>> sorted(["Dog","cat","Rabbit"], key = len)          #按字符串长度排序
['Dog', 'cat', 'Rabbit']
>>> sorted([('Bob',75),('Adam',92),('Lisa',88)])       #默认按元组的第一个元素排序
[('Adam', 92), ('Bob', 75), ('Lisa', 88)]
>>> sorted([('Bob',75),('Adam',92),('Lisa',88)],key = lambda t:t[1])  #按元组的第二个元素排序
[('Bob', 75), ('Lisa', 88), ('Adam', 92)]
```

8.8.10　函数装饰器

Python 函数装饰器(Decorators)使用下列形式装饰一个函数。

```
@装饰器 1
def 函数 1:
    函数体
```

上述定义相当于：

```
函数 1 = 装饰器 1(函数 1)
```

装饰器实际上就是一个函数，一个用来包装函数的函数。装饰器返回一个修改之后的函数对象，且具有相同的函数签名。

装饰器是一种设计模式，其作用是为已经存在的函数添加额外的功能，插入日志以及进行性能测试、事务处理等。

一个函数定义可以使用多个装饰器，结果与装饰器的位置顺序有关。例如：

```
@foo
@spam
def bar(): pass
```

等同于：

```
def bar(): pass
bar = foo(spam(bar))
```

Python 包含内置的装饰器，例如 staticmethod、classmethod 和 property(参见第 9 章)。用户也可以自定义装饰器。

【例 8.50】　函数装饰器示例 1(decorator1.py)。

```
import time, functools
def timeit(func):
    def wrapper( * s):
        start = time.perf_counter()
        func( * s)
        end = time.perf_counter()
        print('运行时间:', end - start)
    return wrapper
@timeit
def my_sum(n):
    sum = 0
    for i in range(n): sum += i
    print(sum)
#测试代码
if __name__ == '__main__':
    my_sum(100000)
```

程序运行结果如下。

```
4999950000
运行时间：0.009712979999999982
```

【例 8.51】 函数装饰器示例 2(decorator2.py)。

```
def makebold(fn):
    def wrappper( * s):
        return "<b>" + fn( * s) + "</b>"
    return wrappper
def makeitalic(fn):
    def wrappper( * s):
        return "<i>" + fn( * s) + "</i>"
    return wrappper
@makebold
@makeitalic
def htmltags(str1):
    return str1
#测试代码
print(htmltags('Hello'))
```

程序运行结果如下。

```
<b><i>Hello</i></b>s
```

8.9 复 习 题

一、选择题

1. Python 语句 print(type(lambda:None))的输出结果是________。

   A. <class 'NoneType'>　　B. <class 'tuple'>
   C. <class 'type'>　　D. <class 'function'>

2. Python 语句序列“f = lambda x,y: x * y; f(12, 34)”的运行结果是________。

   A. 12　　B. 22　　C. 56　　D. 408

3. Python 语句序列“f1=lambda x: x*2; f2=lambda x: x**2; print(f1(f2(2)))”的运行结果是________。

   A. 2　　B. 4　　C. 6　　D. 8

4. 在 Python 中，若有 def f1(p, **p2): print(type(p2))，则 f1(1, a=2)的运行结果是________。

   A. <class 'int'>　　B. <class 'type'>　　C. <class 'dict'>　　D. <class 'list'>

5. 在 Python 中，若有 def f1(a,b,c): print(a+b)，则语句序列“nums=(1,2,3); f1(*nums)”的运行结果是________。

   A. 语法错　　B. 6　　C. 3　　D. 1

二、填空题

1. Python 表达式 eval("5 / 2 + 5 % 2 + 5//2")的结果是________。
2. 如果要为定义在函数外的全局变量赋值，可以使用________语句，表明变量是在外面定义的全局变量。
3. 变量按其作用域大致可以分为________、________和________。
4. 在 Python 的 sys 模块中，函数________和________分别用于获取和设置最大递归次数。

5．在 Python 中，使用内置函数________和________可以查看并输出局部变量和全局变量列表。

三、思考题

1．在 Python 中如何定义一个函数？

2．什么是 Lambda 函数？

3．什么是递归函数？在递归函数的使用过程中为什么需要设置终止条件？

4．下列 Python 语句的输出结果是________。

```
d = lambda p: p * 2; t = lambda p: p * 3
x = 2;x = d(x);x = t(x);x = d(x);print(x)
```

5．下列 Python 语句的输出结果是________。

```
i = map(lambda x: x ** 2, (1, 2, 3))
for t in i: print(t, end = ' ')
```

6．下列 Python 语句的运行结果为________。

```
def f1():
    "simple function"
    pass
print(f1.__doc__)
```

7．下列 Python 语句的输出结果是________。

```
counter = 1;num = 0
def TestVariable():
    global counter
    for i in (1, 2, 3): counter += 1
    num = 10
TestVariable(); print(counter,num)
```

8．下面 Python 程序的功能是什么？输出结果是什么？

```
def f(a, b):
    if b == 0: print(a)
    else: f(b, a % b)
print(f(9, 6))
```

9．下列 Python 语句的输出结果是________。

```
def aFunction():
    "The quick brown fox"
    return 1
print(aFunction.__doc__[4:9])
```

10．下列 Python 语句的输出结果是________。

```
def judge(param1, *param2):
   print(type(param2))
   print(param2)
judge(1, 2, 3, 4, 5)
```

11．下列 Python 语句的输出结果是________。

```
def judge(param1, **param2):
   print(type(param2))
   print(param2)
judge(1, a = 2, b = 3, c = 4, d = 5)
```

8.10 上机实践

1. 完成本章中的例 8.1～例 8.51，熟悉 Python 语言函数和函数式编程。

2. 编写程序，定义一个求阶乘的函数 fact(n)，并编写测试代码，要求输入整数 n(n≥0)。运行效果参见图 8-4。请分别使用递归和非递归方式实现。

3. 编写程序，定义一个求 Fibonacci(斐波那契)数列的函数 fib(n)，并编写测试代码，输出前 20 项(每项宽度 5 个字符位置，右对齐)，每行输出 10 个。运行效果参见图 8-5。请分别使用递归和非递归方式实现。

```
请输入整数n (n>=0) : 5
5 != 120
```

图 8-4 阶乘运行效果

```
    1    1    2    3    5    8   13   21   34   55
   89  144  233  377  610  987 1597 2584 4181 6765
```

图 8-5 斐波那契数列的运行效果

4. 编写程序，利用可变参数定义一个求任意个数数值的最小值的函数 min_n(a, b, *c)，并编写测试代码。例如，对于“print(min_n(8, 2))”以及“print(min_n(16, 1, 7, 4, 15))”的测试代码，程序运行结果如图 8-6 所示。

5. 编写程序，利用元组作为函数的返回值，求序列类型中的最大值、最小值和元素个数，并编写测试代码，假设测试数据分别为 s1＝[9,7,8,3,2,1,55,6]、s2＝["apple","pear","melon","kiwi"]和 s3＝"TheQuickBrownFox"。运行效果参见图 8-7。

```
最小值为 2
最小值为 1
```

图 8-6 利用可变参数求最小值的运行效果

```
list= [9, 7, 8, 3, 2, 1, 55, 6]
最大值= 55 ,最小值= 1 ,元素个数= 8
list= ['apple', 'pear', 'melon', 'kiwi']
最大值= pear ,最小值= apple ,元素个数= 4
list= TheQuickBrownFox
最大值= x ,最小值= B ,元素个数= 16
```

图 8-7 元组作为函数返回值的运行效果

提示：

函数形参为序列类型，返回值是形如“(最大值，最小值，元素个数)”的元组。

8.11 案例研究：井字棋游戏

本章案例研究通过“井字棋游戏”案例帮助读者深入了解使用数据结构和算法实现游戏人工智能。

“井字棋游戏”包括较为复杂的计算机人工智能(AI)落子算法、判断输赢算法等，通过把不同功能定义为独立的函数可以减少程序的复杂性。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第9章

面向对象的程序设计



视频讲解

类是一种数据结构，可以包含数据成员和函数成员。在程序中可以定义类，并创建和使用其对象实例。

9.1 面向对象概念

面向对象的程序设计具有3个基本特征，即封装、继承和多态，可以大大增加程序的可靠性、代码的可重用性和程序的可维护性，从而提高程序的开发效率。

9.1.1 对象的定义

所谓的对象(object)，从概念层面讲，就是某种事物的抽象(功能)。抽象包括数据抽象和过程抽象两个方面：数据抽象就是定义对象的属性；过程抽象就是定义对象的操作。

面向对象的程序设计强调把数据(属性)和操作(服务)结合为一个不可分的系统单位(即对象)，在对象的外部只需要知道它做什么，而不必知道它如何做。

从规格层面讲，对象是一序列可以被其他对象使用的公共接口(对象交互)。从语言实现层面来看，对象封装了数据和代码(数据和程序)。

9.1.2 封装

封装(encapsulation)是面向对象的主要特性。所谓封装，也就是把客观事物抽象并封装成对象，即将数据成员、属性、方法和事件等集合在一个整体内。通过访问控制还可以隐藏内部成员，但只允许可信的对象访问或操作自己的部分数据或方法。

封装保证了对象的独立性，可以防止外部程序破坏对象的内部数据，同时便于对程序的维护和修改。

9.1.3 继承

继承(inheritance)是面向对象的程序设计中代码重用的主要方法。继承允许使用现有类的功能，并在无须重新改写原来类的情况下对这些功能进行扩展。继承可以避免代码复制和相关的代码维护等问题。

9.1.4 多态性

派生类具有基类的所有非私有数据和行为以及新类自己定义的所有其他数据和行为，即子类具有两个有效类型：子类的类型及其继承的基类的类型。对象可以表示多个类型的能力称为多态性(polymorphism)。

多态性允许每个对象以自己的方式去响应共同的消息，从而允许用户以更明确的方式建立通用软件，提高软件开发的可维护性。

9.2 类对象和实例对象

类是一个数据结构，类定义数据类型的数据（属性）和行为（方法）。对象是类的具体实体，也可以称之为类的实例（instance）。

在 Python 语言中，类称为类对象（class object）；类的实例称为实例对象（instance object）。

9.2.1 类对象

类使用关键字 class 声明。类的声明格式如下：

```
class 类名:
    类体
```

其中，类名为有效的标识符，一般为多个单词组成的名称，每个单词除第一个字母大写外，其余的字母均小写；类体由缩进的语句块组成。

定义在类体内的元素都是类的成员。类的主要成员包括两种类型，即描述状态的数据成员（属性）和描述操作的函数成员（方法）。

class 语句实际上是 Python 的复合语句，Python 解释器解释执行 class 语句时会创建一个类对象。

【例 9.1】 创建（定义）类示例（Person1.py）：定义类 Person1，即创建类对象。

```
class Person1:                  #定义类 Person1
    pass                        #类体为空语句
#测试代码
p1 = Person1()                  #创建和使用类对象
print(Person1, type(Person1), id(Person1))
print(p1, type(p1), id(p1))
```

程序运行结果如下。

```
<class '__main__.Person1'> <class 'type'> 1247819394248
<__main__.Person1 object at 0x00000122880690F0> <class '__main__.Person1'> 1247822647536
```

9.2.2 实例对象

类是抽象的，如果要使用类定义的功能，就必须实例化类，即创建类的对象。在创建实例对象后，可以使用“.”运算符来调用其成员。

注意：创建类的对象、创建类的实例、实例化类等说法是等价的，都是以类为模板生成了一个对象。

实例对象的创建和调用格式如下。

```
anObject = 类名(参数列表)
anObject.对象函数 或 anObject.对象属性
```

Python 创建实例对象的方法无须使用关键字 new，而是直接像调用函数一样调用类对象并传递参数，因此类对象是可调用对象（Callable）。在 Python 内置函数中，bool、int、str、list、

dict、set 等均为可调用内置类对象，在有的场合也称之为函数，例如使用 str()函数把数值 123 转换为字符串的形式为 str(123)。

【例 9.2】 实例对象的创建和使用示例。

```
>>> c1 = complex(1, 2)
>>> c1.conjugate()              #输出:(1-2j)
>>> c1.real                     #输出:1.0
```

说明：语句 c1 = complex(1，2)创建类 complex 的实例对象并绑定到变量 c1；表达式 c1.conjugate()调用实例对象 c1 的 conjugate()方法，返回其共轭值(1－2j)；表达式 c1.real 引用实例对象 c1 的实部，返回值 1.0。

9.3 属　　性

类的数据成员是在类中定义的成员变量(域)，用来存储描述类的特征的值，称之为属性。属性可以被该类中定义的方法访问，也可以通过类对象或实例对象进行访问。在函数体或代码块中定义的局部变量只能在其定义的范围内进行访问。

属性实际上是在类中的变量。Python 变量不需要声明，可直接使用。建议用户在类定义的开始位置初始化类属性，或者在构造函数(__init__())中初始化实例属性。

9.3.1 实例对象属性

通过“self.变量名”定义的属性称为实例对象属性，也称为实例对象变量。类的每个实例都包含了该类的实例对象变量的一个单独副本，实例对象变量属于特定的实例。实例对象变量在类的内部通过 self 访问，在外部通过对象实例访问。

实例对象属性一般在__init__()方法中通过如下形式初始化：

```
self.实例变量名 = 初始值
```

然后，在其他实例函数中通过 self 访问：

```
self.实例变量名 = 值                    #写入
self.实例变量名                         #读取
```

或者，在创建对象实例后通过对象实例访问：

```
obj1 = 类名()                           #创建对象实例
obj1.实例变量名 = 值                    #写入
obj1.实例变量名                         #读取
```

【例 9.3】 实例对象属性示例(Person2.py)：定义类 Person2，定义实例属性。

```
class Person2:                          #定义类 Person2
    def __init__(self, name,age):       #__init__()方法
        self.name = name                #初始化 self.name,即成员变量 name(域)
        self.age = age                  #初始化 self.age,即成员变量 age(域)
    def say_hi(self):                   #定义类 Person2 的函数 say_hi()
        print('您好, 我叫', self.name)  #在实例方法中通过 self.name 读取成员变量 name(域)
#测试代码
p1 = Person2('张三',25)                 #创建对象
p1. say_hi()                            #调用对象的方法
print(p1.age)                           #通过 p1.age(obj1.变量名)读取成员变量 age(域)
```

程序运行结果如下。

```
您好, 我叫 张三
25
```

9.3.2 类对象属性

在 Python 中也允许声明属于类对象本身的变量,即类对象属性,也称之为类对象变量、静态属性。类属性属于整个类,不是特定实例的一部分,而是所有实例之间共享一个副本。

类对象属性一般在类体中通过如下形式初始化:

```
类变量名 = 初始值
```

然后,在其类定义的方法中或外部代码中通过类名访问:

```
类名.类变量名 = 值                    #写入
类名.类变量名                         #读取
```

【例 9.4】 类对象属性示例(Person3.py):定义类 Person3,定义类对象属性。

```
class Person3:
    count = 0                        #定义属性 count,表示计数
    name = "Person"                  #定义属性 name,表示名称
#测试代码
Person3.count += 1                   #通过类名访问,将计数加 1
print(Person3.count)                 #类名访问,读取并显示类属性
print(Person3.name)                  #类名访问,读取并显示类属性
p1 = Person3()                       #创建实例对象 1
p2 = Person3()                       #创建实例对象 2
print((p1.name, p2.name))            #通过实例对象访问,读取成员变量的值
Person3.name = "雇员"                #通过类名访问,设置类属性值
print((p1.name, p2.name))            #读取成员变量的值
p1.name = "员工"                     #通过实例对象访问,设置实例对象成员变量的值
print((p1.name, p2.name))            #读取成员变量的值
```

程序运行结果如下。

```
1
Person
('Person', 'Person')
('雇员', '雇员')
('员工', '雇员')
```

说明:类属性如果通过"obj.属性名"来访问,则属于该实例的实例属性。虽然类属性可以使用对象实例来访问,但容易造成困惑,所以建议用户不要这样使用,而是使用标准的访问方式"**类名.类变量名**"。

9.3.3 私有属性和公有属性

Python 类的成员没有访问控制限制,这与其他面向对象的语言不同。

通常约定以两个下画线开头,但是不以两个下画线结束的属性是私有的(private),其他为公共的(public)。注意,不能直接访问私有属性,但可以在方法中访问。

【例 9.5】 私有属性示例(private.py)。

```
class A:
    __name = 'class A'               #私有类属性
```

```
    def get_name():
        print(A.__name)                  #在类方法中访问私有类属性
#测试代码
A.get_name()
A.__name                                 #导致错误,不能直接访问私有类属性
```

程序运行结果如下。

```
class A
Traceback (most recent call last):
   File "C:\pythonpa\ch09\private.py", line 7, in <module>
     A.__name                            #导致错误,不能直接访问私有类属性
AttributeError: type object 'A' has no attribute '__name'
```

9.3.4　@property 装饰器

面向对象编程的封装性原则要求不直接访问类中的数据成员。在 Python 中可以定义私有属性,然后定义相应的访问该私有属性的函数,并使用@property 装饰器来装饰这些函数。程序可以把函数"当作"属性访问,从而提供更加友好的访问方式。

【例 9.6】　property 装饰器示例 1(property1.py)。

```
class Person11:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
#测试代码
p = Person11('王五')
print(p.name)
```

程序运行结果如下。

```
王五
```

@property 装饰器默认提供一个只读属性,如果需要,可以使用对应的 getter、setter 和 deleter 装饰器实现其他访问器函数。

【例 9.7】　property 装饰器示例 2(property2.py)。

```
class Person12:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        self.__name = value
    @name.deleter
    def name(self):
        del self.__name
#测试代码
p = Person12('姚六')
p.name = '王依依'
print(p.name)
```

程序运行结果如下。

```
王依依
```

property 的调用格式如下。

```
property(fget = None, fset = None, fdel = None, doc = None)
```

其中,fget 为 get 访问器;fset 为 set 访问器;fdel 为 del 访问器。

【例 9.8】 property 装饰器示例 3(property3.py)。

```
class Person13:
    def __init__(self, name):
        self.__name = name
    def getname(self):
        return self.__name
    def setname(self, value):
        self.__name = value
    def delname(self):
        del self.__name
    name = property(getname, setname, delname, "I'm the 'name' property.")
#测试代码
p = Person13('爱丽丝');print(p.name)
p.name = '罗伯特'; print(p.name)
```

程序运行结果如下。

```
爱丽丝
罗伯特
```

9.3.5 特殊属性

在 Python 对象中包含许多以双下画线开始和结束的属性,称之为特殊属性。常用的特殊属性如表 9-1 所示。假设示例基于 i=123。

表 9-1 Python 特殊属性

特殊属性	含义	示例
object.__dict__	对象的属性字典	>>> int.__dict__ #mappingproxy({'__repr__': <slot wrapper '__repr__' of 'int' objects>,…
instance.__class__	对象所属的类	>>> i.__class__ #<class 'int'> >>> int.__class__ #<class 'type'>
class.__bases__ class.__base__	类的基类元组 类的基类	>>> int.__bases__ #(<class 'object'>,) >>> int.__base__ #<class 'object'>
class.__name__	类的名称	>>> int.__name__ #'int'
class.__qualname__	类的限定名称	>>> int.__qualname__ #'int'
class.__mro__	方法查找顺序,基类元组	>>> int.__mro__ #(<class 'int'>, <class 'object'>)
class.mro()	同上,可被子类重写	>>> int.mro() #[<class 'int'>, <class 'object'>]
class.__subclasses__()	子类列表	>>> int.__subclasses__() #[<class 'bool'>, <enum 'IntEnum'>, <enum 'IntFlag'>, …

9.3.6　自定义属性

在 Python 中,可以赋予一个对象自定义的属性,即类定义中不存在的属性。对象通过特殊属性__dict__存储自定义属性。例如:

```
>>> class C1:
        pass
>>> o = C1()
>>> o.name = 'custom name'
>>> o.name                              #输出:'custom name'
>>> o.__dict__                          #输出:{'name': 'custom name'}
```

通过重载__getattr__()和__setattr__()可以拦截对成员的访问,从而自定义属性的行为。__getattr__()只有在访问不存在的成员时才会被调用,__getattribute__()拦截所有(包括不存在的成员)的获取操作。在__getattribute__()中不要使用“return self.__dict__[name]”来返回结果,因为在访问 self.__dict__时同样会被__getattribute__()拦截,从而造成无限递归形成死循环。

```
__getattr__(self, name)                 #获取属性,比__getattribute__()优先调用
__getattribute__(self, name)            #获取属性
__setattr__(self, name, value)          #设置属性
__delattr__(self, name)                 #删除属性
```

【例 9.9】　自定义属性示例(custom_attribute.py)。

```
class CustomAttribute(object):
    def __init__(self):
        pass
    def __getattribute__(self, name):
        return str.upper(object.__getattribute__(self, name))
    def __setattr__(self, name, value):
        object.__setattr__(self, name, str.strip(value))
#测试代码
o = CustomAttribute()
o.firstname = '  mary '
print(o.firstname)
```

程序运行结果如下。

```
MARY
```

9.4　方　　法

9.4.1　对象实例方法

方法是与类相关的函数,类方法的定义与普通的函数一致。

在一般情况下,类方法的第一个参数一般为 self,这种方法称为对象实例方法。对象实例方法对类的某个给定的实例进行操作,可以通过 self 显式地访问该实例。对象实例方法的声明格式如下。

```
def 方法名(self,[形参列表]):
    函数体
```

对象实例方法的调用格式如下。

对象.方法名([实参列表])

值得注意的是，虽然类方法的第一个参数为 self，但调用时用户不需要也不能给该参数传值。事实上，Python 自动把对象实例传递给该参数。

例如，假设声明了一个类 MyClass 和类方法 my_func(self,p1,p2)，则：

```
obj1 = MyClass()                      #创建 MyClass 的对象实例 obj1
obj1.my_func (p1,p2)                  #调用对象 obj1 的方法
```

调用对象 obj1 的方法 obj1.my_func(p1,p2)，Python 自动转换为 MyClass.my_func(obj1,p1,p2)，即自动把对象实例 obj1 传值给 self 参数。

注意：Python 中的 self 等价于 C++ 中的 self 指针和 Java、C# 中的 this 关键字。虽然没有限制第一个参数名必须为 self，但建议读者遵循惯例，这样便于阅读和理解，且集成开发环境(IDE)也会提供相应的支持。

【例 9.10】 实例方法示例(PersonMethod.py)：定义类 Person4，创建其对象，并调用对象函数。

```
class Person4:                        #定义类 Person4
    def say_hi(self, name):           #定义方法 say_hi()
        self.name = name              #把参数 name 赋值给 self.name，即成员变量 name(域)
        print('您好, 我叫', self.name)
p4 = Person4()                        #创建对象
p4.say_hi('Alice')                    #调用对象的方法
```

程序运行结果如下。

```
您好, 我叫 Alice
```

9.4.2 静态方法

Python 也允许声明与类的对象实例无关的方法，称之为静态方法。静态方法不对特定实例进行操作，在静态方法中访问对象实例会导致错误。静态方法通过装饰器@staticmethod 来定义，其声明格式如下。

```
@staticmethod
def 静态方法名([形参列表]):
    函数体
```

静态方法一般通过类名来访问，也可以通过对象实例来调用。其调用格式如下。

类名.静态方法名([实参列表])

【例 9.11】 静态方法示例(TemperatureConverter.py)：摄氏温度与华氏温度之间的相互转换。

```
class TemperatureConverter:
    @staticmethod
    def c2f(t_c):                     #摄氏温度到华氏温度的转换
            t_c = float(t_c)
            t_f = (t_c * 9/5) + 32
            return t_f
    @staticmethod
    def f2c(t_f):                     #华氏温度到摄氏温度的转换
            t_f = float(t_f)
```

```
            t_c = (t_f - 32) * 5 /9
            return t_c
#测试代码
print("1. 从摄氏温度到华氏温度.")
print("2. 从华氏温度到摄氏温度.")
choice = int(input("请选择转换方向:"))
if choice == 1:
    t_c = float(input("请输入摄氏温度: "))
    t_f = TemperatureConverter.c2f(t_c)
    print("华氏温度为: {0:.2f}".format(t_f))
elif choice == 2:
    t_f = float(input("请输入华氏温度: "))
    t_c = TemperatureConverter.f2c(t_f)
    print("摄氏温度为: {0:.2f}".format(t_c))
else:
    print("无此选项,只能选择 1 或 2!")
```

程序运行结果如图 9-1 所示。

```
1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向: 1
请输入摄氏温度:  30
华氏温度为:  86.00
```

(a) 从摄氏温度到华氏温度

```
1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向: 2
请输入华氏温度:  70
摄氏温度为:  21.11
```

(b) 从华氏温度到摄氏温度

图 9-1　静态方法示例程序运行结果

9.4.3　类方法

Python 也允许声明属于类本身的方法,即类方法。类方法不对特定实例进行操作,在类方法中访问对象实例属性会导致错误。类方法通过装饰器@classmethod 来定义,第一个形式参数必须为类对象本身,通常为 cls。类方法的声明格式如下。

```
@classmethod
def 类方法名(cls, [形参列表]):
    函数体
```

类方法一般通过类名来访问,也可以通过对象实例来调用。其调用格式如下。

```
类名.类方法名([实参列表])
```

值得注意的是,虽然类方法的第一个参数为 cls,但是调用时用户不需要也不能给该参数传值。事实上,Python 自动把类对象传递给该参数。类对象与类的实例对象不同,在 Python 中类本身也是对象。在调用子类继承父类的类方法时传入的 cls 是子类对象,而非父类对象。

【例 9.12】　类方法示例(classMethod.py)。

```
class Foo:
    classname = "Foo"
    def __init__(self, name):
        self.name = name
    def f1(self):                    #实例方法
        print(self.name)
    @staticmethod
    def f2():                        #静态方法
        print("static")
    @classmethod
```

```
    def f3(cls):                        #类方法
        print(cls.classname)
#测试代码
f = Foo("李")
f.f1()
Foo.f2()
Foo.f3()
```

程序运行结果如下。

```
李
static
Foo
```

9.4.4 __init__()方法和__new__()方法

在 Python 类体中可以定义特殊的方法,例如__new__()方法和__init__()方法。

__new__()方法是一个类方法,在创建对象时调用,返回当前对象的一个实例,一般无须重载该方法。

__init__()方法即构造函数(构造方法),用于执行类的实例的初始化工作。在创建完对象后调用,初始化当前对象的实例,无返回值。

【例 9.13】 __init__()方法示例 1(PersonInit.py)。

```
class Person5:                          #定义类 Person5
    def __init__(self, name):           #__init__()方法
        self.name = name                #把参数 name 赋值给 self.name,即成员变量 name(域)
    def say_hi(self):                   #定义类 Person 的方法 say_hi()
        print('您好, 我叫', self.name)
p5 = Person5('Helen')                   #创建对象
p5.say_hi()                             #调用对象的方法
```

程序运行结果如下。

```
您好, 我叫 Helen
```

【例 9.14】 __init__()方法示例 2(PointInit.py):定义类 Point,表示平面坐标点。

```
class Point:
    def __init__(self, x = 0, y = 0):   #构造函数
        self.x = x
        self.y = y
p1 = Point()                            #创建对象
print("p1({0},{1})".format(p1.x, p1.y))
p1 = Point(5, 5)                        #创建对象
print("p1({0},{1})".format(p1.x, p1.y))
```

程序运行结果如下。

```
p1(0,0)
p1(5,5)
```

9.4.5 __del__()方法

在 Python 类体中可以定义一个特殊的方法——__del__()方法。

__del__()方法即析构函数(析构方法),用于实现销毁类的实例所需的操作,如释放对象占用的非托管资源(例如打开的文件、网络连接等)。

在默认情况下,当对象不再被使用时__del__()方法运行。由于 Python 解释器实现自动垃圾回收,所以无法保证这个方法究竟在什么时候运行。

通过 del 语句可以强制销毁一个对象实例,从而保证调用对象实例的__del__()方法。

【例 9.15】　__del__()方法示例(PersonDel.py)。

```
class Person3:
    count = 0                            #定义类域 count,表示计数
    def __init__(self, name,age):        #构造函数
        self.name = name                 #把参数 name 赋值给 self.name,即成员变量 name(域)
        self.age = age                   #把参数 age 赋值给 self.age,即成员变量 age(域)
        Person3.count += 1               #创建一个实例时计数加 1
    def __del__(self):                   #析构函数
        Person3.count -= 1               #销毁一个实例时计数减 1
    def say_hi(self):                    #定义类 Person3 的方法 say_hi()
        print('您好, 我叫', self.name)
    def get_count():                     #定义类 Person3 的方法 get_count()
        print('总计数为:', Person3.count)
print('总计数为:',Person3.count)          #类名访问
p31 = Person3('张三',25)                  #创建对象
p31.say_hi()                             #调用对象的方法
Person3.get_count()                      #通过类名访问
p32 = Person3('李四',28)                  #创建对象
p32.say_hi()                             #调用对象的方法
Person3.get_count()                      #通过类名访问
del p31                                  #删除对象 p31
Person3.get_count()                      #通过类名访问
del p32                                  #删除对象 p32
Person3.get_count()                      #通过类名访问
```

程序运行结果如下。

```
总计数为: 0
您好, 我叫 张三
总计数为: 1
您好, 我叫 李四
总计数为: 2
总计数为: 1
总计数为: 0
```

9.4.6　私有方法与公有方法

与私有属性类似,Python 约定以两个下画线开头,但不以两个下画线结束的方法是私有的(private),其他为公共的(public)。以双下画线开始和结束的方法是 Python 专有的特殊方法。注意不能直接访问私有方法,但可以在其他方法中访问。

【例 9.16】　私有方法示例(BookPrivate.py)。

```
class Book:                              #定义类 Book
    def __init__(self, name, author, price):
        self.name = name                 #把参数 name 赋值给 self.name,即成员变量 name(域)
        self.author = author             #把参数 author 赋值给 self.author,即成员变量 author(域)
        self.price = price               #把参数 price 赋值给 self.price,即成员变量 price(域)
    def __check_name(self):              #定义私有方法,判断 name 是否为空
        if self.name == '' : return False
        else: return True
```

```
    def get_name(self):                 #定义类 Book 的方法 get_name()
        if self.__check_name():print(self.name,self.author)     #调用私有方法
        else:print('No value')
b = Book('Python 程序设计教程','江红',59.0) #创建对象
b.get_name()                            #调用对象的方法
b.__check_name()                        #直接调用私有方法,非法
```

程序运行结果如下。

```
Python 程序设计教程 江红
Traceback (most recent call last):
  File "C:\pythonpa\ch09\BookPrivate.py", line 14, in <module>
    b.__check_name()                    #直接调用私有方法,非法
AttributeError: 'Book' object has no attribute '__check_name'
```

9.4.7 方法的重载

在其他程序设计语言中方法可以重载,即可以定义多个重名的方法,只要保证方法签名是唯一的即可。方法签名包括 3 个部分,即方法名、参数数量和参数类型。

Python 本身是动态语言,方法的参数没有声明类型(在调用传值时确定参数的类型),参数的数量由可选参数和可变参数来控制。故 Python 对象方法不需要重载,定义一个方法即可实现多种调用,从而实现相当于其他程序设计语言的重载功能。

【例 9.17】 方法重载示例 1(Person21Overload.py)。

```
class Person21:                         #定义类 Person21
    def say_hi(self, name=None):        #定义类方法 say_hi()
        self.name = name                #把参数 name 赋值给 self.name,即成员变量 name(域)
        if name==None: print('您好! ')
        else: print('您好, 我叫', self.name)
p21 = Person21()                        #创建对象
p21.say_hi()                            #调用对象的方法,无参数
p21.say_hi('威尔逊')                    #调用对象的方法,带参数
```

程序运行结果如下。

```
您好!
您好, 我叫 威尔逊
```

在 Python 类体中定义多个重名的方法虽然不会报错,但只有最后一个方法有效,所以建议不要定义重名的方法。

【例 9.18】 方法重载示例 2(Person22Overload.py)。

```
class Person22:                         #定义类 Person22
    def say_hi(self, name):             #定义类方法 say_hi(),带两个参数
        print('您好, 我叫', self.name)
    def say_hi(self, name, age):        #定义类方法 say_hi(),带 3 个参数
        print('hi, {0}, 年龄:{1}'.format(name,age))
p22 = Person22()                        #创建对象
p22.say_hi('Lisa', 22)                  #调用对象的方法
#p22.say_hi('Bob')                      #TypeError: say_hi() missing 1 required positional
                                        #argument: 'age'
```

程序运行结果如下。

```
hi, Lisa,年龄: 22
```

9.5 继　　承

9.5.1 派生类

Python 支持多重继承，即一个派生类可以继承多个基类。派生类的声明格式如下。

```
class 派生类名(基类 1, [基类 2, …]):
    类体
```

其中，派生类名后为所有基类的名称元组。如果在类定义中没有指定基类，则默认其基类为 object。object 是所有对象的根基类，定义了公用方法的默认实现，如__new__()。例如：

```
class Foo: pass
```

等同于：

```
class Foo(object): pass
```

在声明派生类时，必须在其构造函数中调用基类的构造函数。其调用格式如下。

```
基类名.__init__(self, 参数列表)
```

【例 9.19】 派生类示例(DerivedClass.py)：创建基类 Person，它包含两个数据成员 name 和 age；创建派生类 Student，它包含一个数据成员 stu_id。

```
class Person:                                    #基类
    def __init__(self, name, age):               #构造函数
        self.name = name                         #姓名
        self.age = age                           #年龄
    def say_hi(self):                            #定义基类方法 say_hi()
        print('您好, 我叫{0}, {1}岁'.format(self.name,self.age))
class Student(Person):                           #派生类
    def __init__(self, name, age, stu_id):       #构造函数
        Person.__init__(self, name, age)         #调用基类构造函数
        self.stu_id = stu_id                     #学号
    def say_hi(self):                            #定义派生类方法 say_hi()
        Person.say_hi(self)                      #调用基类方法 say_hi()
        print('我是学生, 我的学号为:', self.stu_id)
p1 = Person('张王一', 33)                        #创建对象
p1.say_hi()
s1 = Student('李姚二', 20, '2018101001')         #创建对象
s1.say_hi()
```

程序运行结果如下。

```
您好, 我叫张王一, 33 岁
您好, 我叫李姚二, 20 岁
我是学生, 我的学号为: 2018101001
```

9.5.2 查看继承的层次关系

多个类的继承可以形成层次关系，通过类的方法 mro()或类的属性__mro__可以输出其继承的层次关系。

【例 9.20】 查看类的继承关系示例。

```
>>> class A: pass
```

```
>>> class B(A):pass
>>> class C(B):pass
>>> class D(A):pass
>>> class E(B,D):pass
>>> D.mro()
[<class '__main__.D'>, <class '__main__.A'>, <class 'object'>]
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.B'>, <class '__main__.D'>, <class '__main__.A'>, <class
'object'>)
```

9.5.3 类成员的继承和重写

通过继承,派生类继承基类中除构造方法之外的所有成员。如果在派生类中重新定义从基类继承的方法,则派生类中定义的方法覆盖从基类中继承的方法。

【例 9.21】 类成员的继承和重写示例(SubClass.py)。

```
class Dimension: #定义类 Dimensions
    def __init__(self, x, y):              #构造函数
        self.x = x                         #x 坐标
        self.y = y                         #y 坐标
    def area(self):                        #基类的方法 area()
        pass
class Circle(Dimension):                   #定义类 Circle(圆)
    def __init__(self, r):                 #构造函数
        Dimension.__init__(self, r, 0)
    def area(self):                        #覆盖基类的方法 area()
        return 3.14 * self.x * self.x      #计算圆面积
class Rectangle(Dimension):                #定义类 Rectangle(矩形)
    def __init__(self, w, h):              #构造函数
        Dimension.__init__(self, w, h)
    def area(self):                        #覆盖基类的方法 area()
        return self.x * self.y             #计算矩形面积
d1 = Circle(2.0)                           #创建对象:圆
d2 = Rectangle(2.0, 4.0)                   #创建对象:矩形
print(d1.area(), d2.area())                #计算并打印圆和矩形面积
```

程序运行结果如下。

```
12.56  8.0
```

在该例中,派生类 Circle 和 Rectangle 继承了基类的成员变量 x 和 y,重写了继承的方法 area()。

9.6 对象的特殊方法

9.6.1 对象的特殊方法概述

在 Python 对象中包含许多以双下画线开始和结束的方法,称之为特殊方法。特殊方法通常在针对对象的某种操作时自动调用。

例如,在创建对象实例时(p1 = Person('张三', 23))自动调用其__init__()方法;在解释执行 a<b 时自动调用对象 a 的__lt__()方法。特殊方法如表 9-2 所示。

表 9-2　Python 特殊方法

特殊方法	含　义
__lt__()、__add__()等	对应运算符<、+等
__init__()、__del__()	创建或销毁对象时调用
__len__()	对应于内置函数 len()
__setitem__()、__getitem__()	按索引赋值、取值
__repr__(self)	对应于内置函数 repr()
__str__(self)	对应于内置函数 str()
__bytes__(self)	对应于内置函数 bytes()
__format__(self，format_spec)	对应于内置函数 format()
__bool__(self)	对应于内置函数 bool()
__hash__(self)	对应于内置函数 hash()
__dir__(self)	对应于内置函数 dir()

【例 9.22】　对象的特殊方法示例(SpecialMethod.py)。

```
class Person:
    def __init__(self, name, age):           #特殊方法(构造函数)
        self.name = name
        self.age = age
    def __str__(self):                       #特殊方法,输出成员变量
        return '{0}, {1}'.format(self.name,self.age)
#测试代码
p1 = Person('张三', 23)
print(p1)
```

程序运行结果如下。

```
张三, 23
```

9.6.2　运算符重载与对象的特殊方法

Python 的运算符实际上是通过调用对象的特殊方法实现的。例如：

```
>>> x = 12; y = 23
>>> x + y                                    #等价于调用 x.__add__(y).输出:35
>>> x.__add__(y)                             #输出:35
```

Python 运算符与对应的特殊方法如表 9-3 所示。

表 9-3　运算符与对应的特殊方法

运算符	特殊方法	含　义
<、<=、==、>、>=、!=	__lt__()、__le__()、__eq__() __gt__()、__ge__()、__ne__()	比较运算符
|、^、&	__or__()、__ror__()；__xor__()、__rxor__()；__and__()、__rand__()	按位或、异或、与
|=、^=、&=	__ior__()、__ixor__()、__iand__()	按位复合赋值运算
<<、>>	__lshift__()、__rlshift__()；__rshift__()、__rrshift__()	移位运算
<<=、>>=	__ilshift__()、__irlshift__()；__irshift__()、__irrshift__()	移位复合赋值运算
+、-	__add__()、__radd__()；__sub__()、__rsub__()	加法与减法
+=、-=	__iaddr__()、__isub__()	加减复合赋值运算

续表

运算符	特殊方法	含义
*、/、%、//	__mul__()、__rmul__();__truediv__()、__rtruediv__();__mod__()、__rmod__();__floordiv__()、__rfloordiv__()	乘法、除法、取余、整数除法
*=、/=、%=、//=	__imul__()、__idiv__()、__itruediv__()、__imod__()、__ifloordiv__()	乘除复合赋值运算
+x、-x	__pos__()、__neg__()	正负号
~x	__invert__()	按位翻转
、=	__pow__()、__rpow__()、__ipow__()	指数运算

在 Python 类体中,通过重写各运算符对应的特殊方法即可实现运算符的重载。

【例 9.23】 运算符重载示例(OpOverload.py)。

```
class MyList:                                    #定义类 MyList
    def __init__(self, *args):                   #构造函数
        self.__mylist = []                       #初始化私有属性,空列表
        for arg in args:
            self.__mylist.append(arg)
    def __add__(self, n):                        #重载运算符"+",每个元素增加 n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] += n
    def __sub__(self, n):                        #重载运算符"-",每个元素减少 n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] -= n
    def __mul__(self, n):                        #重载运算符"*",每个元素乘以 n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] *= n
    def __truediv__(self, n):                    #重载运算符"/",每个元素除以 n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] /= n
    def __len__(self):                           #对应于内置函数 len(),返回列表长度
        return(len(self.__mylist))
    def __repr__(self):                          #对应于内置函数 str(),显示列表
        str1 = ''
        for i in range(0, len(self.__mylist)):
            str1 += str(self.__mylist[i]) + ' '
        return str1
#测试代码
m = MyList(1, 2, 3, 4, 5)                        #创建对象
m + 2; print(repr(m))                            #每个元素加 2
m - 1; print(repr(m))                            #每个元素减 1
m * 4; print(repr(m))                            #每个元素乘 4
m / 2; print(repr(m))                            #每个元素除 2
print(len(m))                                    #列表长度
```

程序运行结果如下。

```
3 4 5 6 7
2 3 4 5 6
8 12 16 20 24
4.0 6.0 8.0 10.0 12.0
5
```

9.6.3　@functools.total_ordering 装饰器

支持大小比较的对象需要实现特殊方法,例如__eq__()、__lt__()、__le__()、__ge__()、__gt__(),使用 functools 模块的 total_ordering 装饰器装饰类,则只需要实现__eq__(),以及__lt__()、__le__()、__ge__()、__gt__()中的任意一个。total_ordering 装饰器实现其他比较运算能简化代码量。

【例 9.24】　total_ordering 装饰器函数示例(total_ordering_student.py)。

```
import functools
@functools.total_ordering
class Student:
    def __init__(self, firstname, lastname):  #姓和名
        self.firstname = firstname
        self.lastname = lastname
    def __eq__(self, other):                  #判断姓名是否一致
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):                  #self 姓名<other 姓名
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
#测试代码
if __name__ == '__main__':
    s1 = Student('Mary','Clinton')
    s2 = Student('Mary','Clinton')
    s3 = Student('Charlie','Clinton')
    print(s1 == s2)
    print(s1 > s3)
```

程序运行结果如下。

```
True
True
```

9.6.4　__call__()方法和可调用对象

在 Python 类体中可以定义一个特殊的方法——__call__()方法。定义了__call__()方法的对象称为可调用对象(callable),即该对象可以像函数一样被调用。

【例 9.25】　可调用对象示例(CallableObj.py)。

```
class GDistance:                                #类:自由落体距离
    def __init__(self, g):                      #构造函数
        self.g = g
    def __call__(self, t):                      #自由落体下落距离
        return (self.g * t * * 2)/2
#测试代码
if __name__ == '__main__':
    e_gdist = GDistance(9.8)                    #地球上的重力加速度
    for t in range(11):                         #自由落体 0~10 秒的下落距离
        print(format(e_gdist(t), "0.2f"),end = ' ')  #调用可调用对象 e_gdist
```

程序运行结果如下。

```
0.00 4.90 19.60 44.10 78.40 122.50 176.40 240.10 313.60 396.90 490.00
```

9.7 对象的引用、浅拷贝和深拷贝

9.7.1 对象的引用

对象的赋值实际上是对象的引用,在创建一个对象并把它赋值给一个变量时,该变量是指向该对象的引用,其 id()返回值保持一致。

【例 9.26】 对象的引用示例。若银行卡采用列表[户主名,[卡种别,金额]]表示,则:

```
>>> acc10 = ['Charlie', ['credit', 0.0]]        #创建列表对象(信用卡账户),变量 acc10 代表主卡
>>> acc11 = acc10                               #变量 acc11 代表副卡,指向 acc10(主卡)的对象
>>> id(acc10),id(acc11)                         #二者 id 相同:(2739033039112, 2739033039112)
```

9.7.2 对象的浅拷贝

对象的赋值引用同一个对象,即不复制对象。如果要复制对象,可以使用下列方法之一。

- 切片操作:例如 acc11[:]。
- 对象实例化:例如 list(acc11)。
- copy 模块的 copy()函数:例如 copy.copy(acc1)。

【例 9.27】 对象的浅拷贝示例。

```
>>> import copy
>>> acc1 = ['Charlie', ['credit', 0.0]]
>>> acc2 = acc1[:]                              #使用切片方式复制对象
>>> acc3 = list(acc1)                           #使用对象实例化方法复制对象
>>> acc4 = copy.copy(acc1)                      #使用 copy.copy()函数复制对象
>>> id(acc1),id(acc2),id(acc3),id(acc4)         #复制对象 id 各不相同
(2739033039240, 2739033040008, 2739035724168, 2739033039880)
>>> acc2[0] = 'Mary'                            #acc2 的第 1 个元素赋值,即户主为'Mary'
>>> acc2[1][1] = -99.9                      #acc2 的第 2 个元素的第 2 个元素赋值,即消费金额为 99.9
>>> acc1, acc2                                  #注意,acc2 消费金额改变为 99.9,acc1 也随之改变
(['Charlie', ['credit', -99.9]], ['Mary', ['credit', -99.9]])
>>> id(acc1[1]),id(acc2[1])                     # acc1[1]和 acc2[1]指向同一个对象
(2739033038152, 2739033038152)
```

在该例中,acc2[1][1]赋值-99.9,acc1[1][1]也一同改变,因为二者指向同一个对象。

Python 复制一般是浅拷贝,即复制对象时对象中包含的子对象并不复制,而是引用同一个子对象。如果要递归复制对象中包含的子对象,请参见 9.7.3 节。

9.7.3 对象的深拷贝

如果要递归复制对象中包含的子对象,可以使用 copy 模块的 deepcopy()函数。

【例 9.28】 对象的深拷贝示例。

```
>>> import copy
>>> acc1 = ['Charlie', ['credit', 0.0]]
>>> acc5 = copy.deepcopy(acc1)                  #使用 copy.deepcopy()函数深拷贝对象
>>> acc5[0] = 'Clinton'                         #acc5 的第 1 个元素赋值,即户主为'Clinton'
>>> acc5[1][1] = -19.9                          #acc5 的第 2 个元素的第 2 个元素赋值,即消费金额为
                                                #19.9
>>> acc1,acc5                      # (['Charlie', ['credit', 0.0]], ['Clinton', ['credit', -19.9]])
```

```
>>> id(acc1),id(acc5),id(acc1[1]),id(acc5[1])
(2739033040648, 2739033040264, 2739033040520, 2739033039688)
```

9.8　可迭代对象：迭代器和生成器

可循环迭代的对象称为可迭代对象，迭代器和生成器函数是可迭代对象，在 Python 中提供了定义迭代器和生成器的协议和方法。

相对于序列，可迭代对象仅在迭代时产生数据，故可以节省内存空间。Python 语言提供了若干内置可迭代对象，例如 range、map、filter、enumerate、zip；在标准库 itertools 模块中包含各种迭代器，这些迭代器非常高效，且内存消耗小。迭代器既可以单独使用，也可以组合使用。

9.8.1　可迭代对象

在 Python 中，实现了__iter__()的对象是可迭代对象。在 collections.abc 模块中定义了抽象基类 Iterable，使用内置的 isinstance()可以判断一个对象是否为可迭代对象。序列对象都是可迭代对象，生成器函数和生成器表达式也是可迭代对象。例如：

```
>>> import collections.abc
>>> isinstance((1,2,3),collections.abc.Iterable)        #True
>>> isinstance('python33',collections.abc.Iterable)     #True
>>> isinstance(123,collections.abc.Iterable)            #False
```

9.8.2　迭代器

实现了__next__()的对象是迭代器，可以使用内置函数 next()调用迭代器的__next__()方法依次返回下一个项目值，如果没有新项目，则将导致 StopIteration。

在 collections.abc 模块中定义了抽象基类 Iterator。使用迭代器可以实现对象的迭代循环，迭代器让程序更加通用、优雅、高效，更加 Python 化。对于大量项目的迭代，使用列表会占用更多的内存，而使用迭代器可以避免之。例如：

```
>>> import collections.abc
>>> i1 = (i**2 for i in range(10))
>>> isinstance(i1, collections.abc.Iterator) #True
```

9.8.3　迭代器协议

迭代器对象必须实现两个方法，即__iter__()和__next__()，二者合称为迭代器协议。__iter__()用于返回对象本身，以方便 for 语句进行迭代；__next__()用于返回下一元素。例如：

```
>>> i1 = (i**2 for i in range(10))
>>> help(i1)
...
| __iter__(...)
|     x.__iter__() <==> iter(x)
| __next__(...)
|     x.__next__() <==> next(x)
...
```

9.8.4 可迭代对象的迭代：iter()函数和 next()函数

使用内置函数 iter(obj)可以调用可迭代对象 obj 的__iter__()方法，以返回一个迭代器(iterator)。

使用内置函数 iter(iterable)可以返回可迭代对象 iterable 的迭代器；使用内置函数 next()可以依次返回迭代器对象的下一个项目值，如果没有新项目，则将导致 StopIteration。例如：

```
>>> t = (1,2)                             #元组
>>> i = iter(t)                           #通过内置函数 iter()获得 iterator
>>> next(i)                               #通过内置函数 next()获得下一个项目:1
>>> next(i)                               #通过内置函数 next()获得下一个项目:2
>>> next(i)                               #没有新项目时将导致 StopIteration
```

使用 while 循环也可以循环迭代可迭代对象。

【例 9.29】 使用 while 循环迭代可迭代对象(while.py)。

```
t = (1,2,3,4,5,6,7,8,9,0)                 #元组
fetch = iter(t)                           #获取迭代器
while True:
    try: i = next(fetch)
    except StopIteration: break
    print(i, end = ' ')
```

程序运行结果如下。

```
1 2 3 4 5 6 7 8 9 0
```

9.8.5 可迭代对象的迭代：for 语句

通常使用 for 语句实现可迭代对象的迭代。Python 中的 for 循环实现了自动迭代可迭代对象的功能。例如：

```
>>> i1 = (1,2,3,4,5,6,7,8,9,0)            #元组
>>> for i in i1: print(i, end = ' ')      #1 2 3 4 5 6 7 8 9 0
>>> i2 = [1,2,3,4,5,6,7,8,9,0]            #列表
>>> for i in i2: print(i, end = ' ')      #1 2 3 4 5 6 7 8 9 0
>>> i3 = 'python33'                       #字符串
>>> for i in i3: print(i, end = ' ')      #p y t h o n 3 3
>>> i4 = range(10)                        #可迭代对象
>>> for i in i4: print(i, end = ' ')      #0 1 2 3 4 5 6 7 8 9
```

9.8.6 自定义可迭代对象和迭代器

声明一个类，定义__iter__()方法和__next__()方法。创建该类的对象，既是可迭代对象，也是迭代器。

【例 9.30】 定义类 Fib，实现 Fibonacci 数列(fibonacciIterNext.py)。Fib 对象定义了__iter__()方法和__next__()方法，所以是可迭代对象，也是迭代器。

```
class Fib:
    def __init__(self):
        self.a, self.b = 0,1              #前两项值
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a                     #f(n) = f(n-1) + f(n-2)
```

```
    def __iter__(self):
        return self
#测试代码
fibs = Fib()
for f in fibs:
    if f < 1000: print(f, end = ',')
    else: break
```

程序运行结果如下。

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

9.8.7　生成器函数

在函数定义中，如果使用 yield 语句代替 return 返回一个值，则定义了一个生成器函数(generator)。

生成器函数使用 yield 语句返回一个值，然后保存当前函数的整个执行状态，等待下一次调用。生成器函数是一个迭代器，是可迭代对象，支持迭代。例如：

```
>>> def gentripls(n):
        for i in range(n):
            yield i * 3
>>> f = gentripls(10)
>>> f                              # <generator object gentripls at 0x000001ED6C57DA20>
>>> i = iter(f)                    #通过内置函数 iter()获得 iterator
>>> next(i)                        #通过内置函数 next()获得下一个项目:0
>>> next(i)                        #通过内置函数 next()获得下一个项目:3
>>> for t in f: print(t, end = ' ')    #6 9 12 15 18 21 24 27
```

【例 9.31】　利用生成器函数创建 Fibonacci 数列(fibonacci_yield.py)。

```
def fib():
    a,b = 0,1                      #前两项值
    while 1:
        a,b = b,a + b
        yield a                    #f(n) = f(n - 1) + f(n - 2)
#测试代码
if __name__ == '__main__':
    fibs = fib()
    for f in fibs:
        if f < 1000: print(f, end = ',')
        else: break
```

程序运行结果如下。

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

【例 9.32】　利用生成器函数创建返回 m 到 n 之间素数的生成器(primes_yield.py)。

```
import math
def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
```

```
    return True
def primes(m, n):
    """返回[m, n]的所有素数的生成器"""
    for i in range(m, n+1):
        if is_prime(i):
            yield i
#测试代码
if __name__ == '__main__':
    pimes1 = primes(5000000000, 5000000090)
    #pimes1 = primes(1, 100)
    for p in pimes1:
        print(p, end=',')
```

程序运行结果如下。

```
5000000029,5000000039,5000000059,5000000063,
```

9.8.8 反向迭代：reversed 迭代器

使用内置函数 reversed()可以实现一个序列的反向序列。如果一个可迭代对象实现了__reversed__()方法，则可以使用 reversed()函数获得其反向可迭代对象。

只有长度有限的序列或者实现了__reversed__()方法的可迭代对象才可以使用内置函数 reversed()。例如：

```
>>> reversed([1, 2, 3, 4, 5])          #<list_reverseiterator object at 0x000001ED6C5080F0>
>>> for i in reversed([1, 2, 3, 4, 5]): print(i, end=' ')    #5 4 3 2 1
```

【例 9.33】 可反向迭代的迭代器示例(reversedCountdown.py)。

```
class Countdown:
    def __init__(self, start):
        self.start = start
    #正向迭代
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1
    #反向迭代
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
#测试代码
if __name__ == '__main__':                  #如果独立运行，则运行测试代码
    for i in Countdown(10): print(i, end=' ')
    for i in reversed(Countdown(10)): print(i, end=' ')
```

程序运行结果如下。

```
10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10
```

9.8.9 生成器表达式

使用生成器表达式可以简便、快捷地返回一个生成器。生成器表达式的语法和列表解析基本一样，只不过生成器表达式使用()代替[]。

生成器表达式的形式如下：

```
(expr for iter_var in iterable)                  #迭代 iterable 的所有内容,计算并返回生成器
(expr for iter_var in iterable if cond_expr)     #按条件迭代,计算并返回生成器
```

表达式 expr 使用每次迭代的内容 iter_var 计算生成一个列表。如果指定了条件表达式 cond_expr,则只有满足条件的 iterable 元素参与迭代。例如：

```
>>> (i ** 2 for i in range(10))                  #<generator object <genexpr> at 0x000001ED6C57DA98>
>>> for j in (i ** 2 for i in range(10)):
        print(j, end = ' ')                      #0 1 4 9 16 25 36 49 64 81
>>> for j in (i for i in range(10) if i % 2 == 0):
        print(j, end = ' ')                      #0 2 4 6 8
```

9.8.10　range 可迭代对象

Python 3 中的 range 等同于 Python 2.x 中的 xrange,Python 2.x 中的 range()函数直接在内存中生成数字列表。Python 3 中的 range 是一个可迭代对象,在迭代时产生指定范围的数字序列,故可以节省内存空间。

```
range(start, stop[, step])                       #构造函数
```

【例 9.34】　range 可迭代对象示例。

```
>>> range                                        #<class 'range'>
>>> for i in range(1, 10): print(i, end = ',')   #循环输出可迭代对象的内容:1,2,3,4,5,6,7,8,9,
>>> list(range(1,10,2))                          #把可迭代对象转换为列表输出:[1, 3, 5, 7, 9]
```

9.8.11　map 迭代器和 itertools.starmap 迭代器

Python 3 中的 map 是可迭代对象,使用指定函数处理可迭代对象的每个元素(如果函数需要多个参数,则对应各可迭代对象),返回结果可迭代对象。

```
map(function, iterable, …)                       #构造函数
```

【例 9.35】　map 迭代器示例。

```
>>> map                                          #<class 'map'>
>>> list(map(abs, (1, -2, 3)))                   #[1, 2, 3]
>>> import operator
>>> list(map(operator.add, (1, 2, 3), (1, 2, 3)))  #[2, 4, 6]
```

如果函数的参数为元组,则需要使用 itertools.starmap 迭代器：

```
itertools.starmap(function, iterable)            #构造函数
```

【例 9.36】　itertools.starmap 迭代器示例。

```
>>> import itertools
>>> list(itertools.starmap(pow, [(2,5), (3,2), (10,3)]))    #[32, 9, 1000]
```

9.8.12　filter 迭代器和 itertools.filterfalse 迭代器

Python 3 中的 filter 是可迭代对象,使用指定函数处理可迭代对象的每个元素,函数返回 bool 类型的值。若结果为 True,则返回该元素。如果 function 为 None,则返回元素为 True 的元素。

```
filter(function, iterable)                       #构造函数
```

【例 9.37】 filter 迭代器示例。

```
>>> filter                                          #<class 'filter'>
>>> list(filter(lambda x: x>0, (-1, 2, -3, 0, 5)))  #[2, 5]
>>> list(filter(None, (1, 2, 3, 0, 5)))             #[1, 2, 3, 5]
```

如果需要返回结果为 False 的元素,则需要使用 itertools. filterfalse 迭代器:

```
filterfalse(predicate, iterable)                    #构造函数
```

filterfalse 根据条件函数 predicate 处理可迭代对象的每个元素,若结果为 True,则丢弃,否则返回该元素。例如:

```
>>> import itertools
>>> list(itertools.filterfalse(lambda x: x%2, range(10))) #[0, 2, 4, 6, 8]
```

9.8.13 zip 迭代器和 itertools. zip_longest 迭代器

Python 3 中的 zip 是可迭代对象,用于拼接多个可迭代对象 iter1、iter2……的元素,返回新的可迭代对象,其元素为各序列 iter1、iter2……对象元素组成的元组。如果各序列 iter1、iter2…的长度不一致,则截至最小序列长度,这样可以节省内存空间。

```
zip(*iterables)                        #构造函数
```

【例 9.38】 zip 迭代器示例。

```
>>> zip                                #<class 'zip'>
>>> zip((1,2,3),'abc', range(3))       # <zip object at 0x000001ED6C5A72C8>
>>> list(zip((1,2,3),'abc', range(3))) #[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2)]
>>> list(zip('abc', range(10)))        #[('a', 0), ('b', 1), ('c', 2)]
```

若多个可迭代对象的元素个数不一致,如果需要取最大的长度,则需要使用 itertools. zip_longest 迭代器:

```
zip_longest(*iterables, fillvalue=None)
```

其中,fillvalue 是填充值,默认为 None。

【例 9.39】 itertools. zip_longest 迭代器示例。

```
>>> import itertools
>>> list(itertools.zip_longest('ABCD', 'xy', fillvalue='-'))
[('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

9.8.14 enumerate 迭代器

Python 3 中的 enumerate 是可迭代对象,用于枚举可迭代对象 iterable 中的元素,返回元素为元组(计数, 元素)的可迭代对象。注意,计数从 start 开始(默认为 0)。

```
enumerate(iterable, start=0)           #构造函数
```

【例 9.40】 enumerate 迭代器示例 1。

```
>>> enumerate                          #<class 'enumerate'>
>>> list(enumerate('ABCD', start=10001))
[(10001, 'A'), (10002, 'B'), (10003, 'C'), (10004, 'D')]
```

【例 9.41】 enumerate 迭代器示例 2(enumerate_lineno. py):打印文本文件的行号和内容。

```
def printfilewithlineno(path):
    with open(path, 'r', encoding = 'utf8') as f:
        lines = f.readlines()
    for idx, line in enumerate(lines):
        print(idx, line)
#测试代码
if __name__ == '__main__':
    thisfile = __file__
printfilewithlineno(thisfile)
```

程序运行结果如下。

```
0 def printfilewithlineno(path):
1 with open(path, 'r') as f:
...
```

9.8.15　无穷序列迭代器 itertools.count、cycle 和 repeat

itertools 模块包含 3 个无穷序列的迭代器：

- **count(start = 0, step = 1)**　　# 从 start 开始,步长为 step 的无穷序列
- **cycle(iterable)**　　# 可迭代对象 iterable 元素的无限重复
- **repeat(object[, times])**　　# 重复对象 object 无数次(若指定 times,则重复 times 次)

【例 9.42】　无穷序列迭代器 itertools.count、cycle 和 repeat 示例。

```
>>> from itertools import *
>>> list(zip(count(1), 'abcde'))          #[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
>>> list(zip(range(10), cycle('abc')))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'a'), (4, 'b'), (5, 'c'), (6, 'a'), (7, 'b'), (8, 'c'), (9, 'a')]
>>> list(repeat('God', 5))                #['God', 'God', 'God', 'God', 'God']
```

9.8.16　累计迭代器 itertools.accumulate

itertools 模块的 accumulate 迭代器用于返回累计和：

accumulate(iterable[, func])

其中,若可迭代对象 iterable 的元素 p0、p1、p2……为数值,则结果为 p0、p0+p1、p0+p1+p2……。如果指定了带两个参数的 func,则 func 代替默认的加法运算。

【例 9.43】　累计迭代器 itertools.accumulate 示例。

```
>>> import itertools
>>> list(accumulate((1,2,3,4,5)))          #结果元素为 1、1+2、1+2+3、……,即[1, 3, 6, 10, 15]
>>> list(accumulate((1,2,3,4,5), operator.mul))  #使用乘法作为运算,结果为[1, 2, 6, 24, 120]
```

9.8.17　级联迭代器 itertools.chain

itertools 模块的 chain 迭代器用于返回级联元素：

chain(* iterables)　　# 构造函数

其用于连接所有的可迭代对象 iterables1、iterables2……即连接多个可迭代对象的元素,作为一个序列。chain 的类工厂函数 chain.from_iterable(iterable)也可以用于连接多个序列。

【例 9.44】　级联迭代器 chain 示例。

```
>>> import itertools
>>> list(itertools.chain((1,2,3),'abc',range(5)))        #[1, 2, 3, 'a', 'b', 'c', 0, 1, 2, 3, 4]
>>> list(itertools.chain.from_iterable(['ABC', 'DEF']))  #['A', 'B', 'C', 'D', 'E', 'F']
```

9.8.18 选择压缩迭代器 itertools.compress

itertools 模块的 compress 迭代器用于返回可迭代对象的部分元素:

```
compress(data, selectors)                           #构造函数
```

其根据选择器 selectors 的元素(True/False),返回元素为 True 对应的 data 序列中的元素。当 data 序列或 selectors 终止时停止判断。

【例 9.45】 选择压缩迭代器 itertools.compress 示例。

```
>>> import itertools
>>> list(itertools.compress('ABCDEF', [1,0,1,0,1,1]))     #['A', 'C', 'E', 'F']
```

9.8.19 截取迭代器 itertools.dropwhile 和 takewhile

itertools 模块的 dropwhile 和 takewhile 迭代器用于返回可迭代对象的部分元素:

```
dropwhile(predicate, iterable)                      #构造函数
takewhile(predicate, iterable)                      #构造函数
```

dropwhile 根据条件函数 predicate 处理可迭代对象的每个元素,丢弃 iterable 的元素,直到条件函数的结果为 True;takewhile 则根据条件函数 predicate 处理可迭代对象的每个元素,返回 iterable 的元素,直到条件函数的结果为 False。

【例 9.46】 截取迭代器 itertools.dropwhile 和 takewhile 示例。

```
>>> import itertools
>>> list(itertools.dropwhile(lambda x: x<5, [1,4,6,4,1]))    #[6, 4, 1]
>>> list(itertools.takewhile(lambda x: x<5, [1,4,6,4,1]))    #[1, 4]
```

9.8.20 切片迭代器 itertools.islice

itertools 模块的 islice 迭代器用于返回可迭代对象的切片:

```
islice(iterable, stop)                              #构造函数
islice(iterable, start, stop[, step])               #构造函数
```

序列支持切片操作,同样,可迭代对象可使用 islice 实现切片功能。islice 返回可迭代对象 iterable 的切片,从索引位置 start(第 1 个元素为 0)开始到 stop(不包括)结束,步长为 step(默认为 1)。如果 stop 为 None,则操作直到结束。

【例 9.47】 切片迭代器 itertools.islice 示例。

```
>>> import itertools
>>> list(itertools.islice('ABCDEFG', 2))              #['A', 'B']
>>> list(itertools.islice('ABCDEFG', 2, 4))           #['C', 'D']
>>> list(itertools.islice('ABCDEFG', 2, None))        #['C', 'D', 'E', 'F', 'G']
>>> list(itertools.islice('ABCDEFG', 0, None, 2))     #['A', 'C', 'E', 'G']
```

9.8.21 分组迭代器 itertools.groupby

itertools 模块的 groupby 迭代器用于返回可迭代对象的分组:

```
groupby(iterable, key=None)                         #构造函数
```

其中,iterable 为待分组的可迭代对象;可选的 key 为用于计算键值的函数,默认为 None,即键值为元素本身值。groupby 返回的结果为迭代器,其元素为(key, group),其中 key 是分组

的键值，group 为 iterable 中具有相同 key 值的元素的集合的子迭代器。

【例 9.48】　分组迭代器 itertools.groupby 示例。

```
>>> import itertools
>>> data = [1, -2,0,0, -1,2,1, -1,2,0,0]; data1 = sorted(data, key = abs)
>>> for k, g in itertools.groupby(data1, key = abs):
        print(k, list(g))
0 [0, 0, 0, 0]
1 [1, -1, 1, -1]
2 [-2, 2, 2]
```

9.8.22　返回多个迭代器 itertools.tee

itertools 模块的 tee 迭代器用于返回多个可迭代对象：

```
tee(iterable, n = 2)                                    #构造函数
```

其返回可迭代对象 iterable 的 n 个(默认为 2)迭代器。例如：

```
>>> import itertools
>>> for i in itertools.tee(range(10), 3): print(list(i))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

9.8.23　组合迭代器 itertools.combinations 和 combinations_with_replacement

itertools 模块的 combinations(元素不重复)和 combinations_with_replacement(元素可重复)迭代器用于序列的组合：

- **combinations(iterable, r)**　　　　#构造函数
- **combinations_with_replacement(iterable, r)**　　　　#构造函数

其返回可迭代对象 iterable 的元素的组合，组合长度为 r。

【例 9.49】　组合迭代器 itertools.combinations 和 combinations_with_replacement 示例。

```
>>> import itertools
>>> list(itertools.combinations([1,2,3],2))   #[(1, 2), (1, 3), (2, 3)]
>>> list(itertools.combinations([1,2,3,4],2))#[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
>>> list(itertools.combinations([1,2,3,4],3))#[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
>>> list(itertools.combinations_with_replacement([1,2,3],2))
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

9.8.24　排列迭代器 itertools.permutations

itertools 模块的 permutations 迭代器用于序列的排列：

```
permutations(iterable, r = None)                #构造函数
```

其返回可迭代对象 iterable 的元素的排列，组合长度为 r(默认为序列长度)。

【例 9.50】　排列迭代器 itertools.permutations 示例。

```
>>> import itertools
>>> list(itertools.permutations([1,2,3],2))    #[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(itertools.permutations([1,2,3]))
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

9.8.25 笛卡儿积迭代器 itertools.product

itertools 模块的 product 迭代器用于序列的笛卡儿积:

```
product( * iterables, repeat = 1)                    # 构造函数
```

其返回可迭代对象 iterables1、iterables2……的元素的笛卡儿积,repeat 为可迭代对象的重复次数(默认为 1)。

【例 9.51】 笛卡儿积迭代器 itertools.product 示例。

```
>>> import itertools
>>> list(itertools.product([1,2],'abc'))
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c')]
>>> list(itertools.product([1,2],repeat = 3))
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
```

9.9 自定义类应用举例

在 Python 语言、标准库和第三方库中定义了大量的类,类是 Python 语言的主要数据结构。用户也可以通过自定义类创建和使用新的数据结构。

9.9.1 Color 类

Color 类封装使用 RGB 颜色模型表示颜色及相应功能。Color 类的设计思路如下:

(1) 定义带 3 个 0 到 255 的整数参数 r、g、b 的构造函数,用于初始化对应于红、绿、蓝 3 种颜色分量的实例对象属性_r、_g 和_b。

(2) 通过装饰器@property 定义 3 个可以作为属性访问的实例对象方法 r()、g()和 b()。

(3) 定义用于计算颜色亮度的方法 luminance(self):$Y = 0.299r + 0.587g + 0.114b$。

(4) 定义用于转换为灰度颜色亮度的方法 toGray(self)。

(5) 定义用于比较两种颜色兼容性的方法 isCompatible(self, c)。颜色兼容性指在以一种颜色为背景时另一种颜色的可阅读性。一般而言,前景色和背景色的亮度差至少应该是 128。例如,白纸黑字的亮度差为 255。

【例 9.52】 实现 RGB 颜色模型的 Color 类(color.py)。

```
class Color:
    """表示 RGB 模型的类"""
    def __init__(self, r = 0, g = 0, b = 0):
        """构造函数"""
        self._r = r                                 #Red(红色)分量
        self._g = g                                 #Green(绿色)分量
        self._b = b                                 #Blue(蓝色)分量
    @property
    def r(self):
        return self._r
    @property
    def g(self):
        return self._g
```

```
    @property
    def b(self):
        return self._b
    def luminance(self):
        """计算并返回颜色的亮度"""
        return .299 * self._r + .587 * self._g + .114 * self._b
    def toGray(self):
        """转换为灰度颜色"""
        y = int(round(self.luminance()))
        return Color(y, y, y)
    def isCompatible(self, c):
        """比较前景色和背景色是否匹配"""
        return abs(self.luminance() - c.luminance()) >= 128.0
    def __str__(self):
        """重载方法,输出:(r, g, b)"""
        return '({},{},{})'.format(self._r,self._g,self._b)
#常用颜色
WHITE = Color(255, 255, 255)
BLACK = Color( 0, 0, 0)
RED = Color(255, 0, 0)
GREEN = Color( 0, 255, 0)
BLUE = Color( 0, 0, 255)
CYAN = Color( 0, 255, 255)
MAGENTA = Color(255, 0, 255)
YELLOW = Color(255, 255, 0)
#测试代码
if __name__ == '__main__':
    c = Color(255, 200, 0)                                       #ORANGE(橙色)
    print('颜色字符串:{}'.format(c))                              #输出颜色字符串
    print('颜色分量:r={},g={},b={}'.format(c.r, c.g, c.b))        #输出各颜色分量
    print('颜色亮度:{}'.format(c.luminance()))                    #输出颜色亮度
    print('转换为灰度颜色:{}'.format(c.toGray()))                  #输出转换后的灰度颜色
    print('{}和{}是否匹配:{}'.format(c,RED,c.isCompatible(RED)))  #比较与红色是否匹配
```

程序运行结果如下。

```
颜色字符串:(255,200,0)
颜色分量:r=255,g=200,b=0
颜色亮度:193.64499999999998
转换为灰度颜色:(194,194,194)
(255,200,0)和(255,0,0)是否匹配:False
```

9.9.2 Histogram 类

Histogram 类封装直方图(包括数据及基本统计功能)。Histogram 类的设计思路如下:

(1) 定义带一个整数参数 n 的构造函数,用于初始化存储数据的列表,列表长度为 n,列表各元素的初始值为 0。

(2) 定义实例对象方法 addDataPoint(self, i),用于增加一个数据点。

(3) 定义用于计算数据点个数之和、平均值、最大值、最小值的实例对象方法,即 count()、mean()、max()、min()。

(4) 定义用于绘制简单直方图的实例对象方法 draw()。

【例 9.53】 实现直方图类 Histogram(histogram.py)。

```
import random
import math
class Stat:
    def __init__(self, n):
        self._data = []
        for i in range(n):
            self._data.append(0)
    def addDataPoint(self, i):
        """增加数据点"""
        self._data[i] += 1
    def count(self):
        """计算数据点个数之和(统计数据点个数)"""
        return sum(self._data)
    def mean(self):
        """计算各数据点个数的平均值"""
        return sum(self._data)/len(self._data)
    def max(self):
        """计算各数据点个数的最大值"""
        return max(self._data)
    def min(self):
        """计算各数据点个数的最小值"""
        return min(self._data)
    def draw(self):
        """绘制简易直方图"""
        for i in self._data:
            print('#'* i)
#测试代码
if __name__ == '__main__':
    #随机生成 100 个的 0 到 9 的数
    st = Stat(10)
    for i in range(100):
        score = random.randrange(0,10)
        st.addDataPoint(math.floor(score))
    print('数据点个数:{}'.format(st.count()))
    print('数据点个数的平均值:{}'.format(st.mean()))
    print('数据点个数的最大值:{}'.format(st.max()))
    print('数据点个数的最小值:{}'.format(st.min()))
    st.draw()        #绘制简易直方图
```

程序运行结果(随机生成,每次运行结果不同)如图 9-2 所示。

```
数据点个数: 100
数据点个数的平均值: 10.0
数据点个数的最大值: 17
数据点个数的最小值: 5
########
##########
#########
########
#######
##############
#################
#####
################
######
```

图 9-2 直方图 Histogram 类运行效果

9.10　复　习　题

一、填空题

1. 面向对象的程序设计具有 3 个基本特征，即________、________和________。
2. Python 语句序列“x='123'; print(isinstance(x, int))”的运行结果为________。
3. 在 Python 中创建对象后可以使用________运算符来调用其成员。
4. 在 Python 类体中，________是一个类方法，在创建对象时调用，返回当前对象的一个实例，一般无须重载该方法。________方法即构造函数（构造方法），用于执行类的实例的初始化工作，在对象创建后调用，初始化当前对象的实例，无返回值。________方法即析构函数，用于实现销毁类的实例所需的操作，例如释放对象占用的非托管资源。
5. 在 Python 中，实例变量在类的内部通过________访问，在外部通过对象实例访问。

二、思考题

1. Python 如何复制一个对象？
2. Python 提供了哪些特殊属性？如何表示这些特殊属性？它们各自的含义是什么？
3. 下列 Python 语句的程序运行结果为________。

```
class parent:
    def __init__(self, param):
        self.v1 = param
class child(parent):
    def __init__(self, param):
        parent.__init__(self, param)
        self.v2 = param
obj = child(100); print ("%d %d" % (obj.v1, obj.v2))
```

4. 下列 Python 语句的程序运行结果为________。

```
class Account:
    def __init__(self, id):
        self.id = id; id = 888
acc = Account(100); print(acc.id)
```

5. 下列 Python 语句的程序运行结果为________。

```
class account:
    def __init__(self, id, balance):
        self.id = id; self.balance = balance
    def deposit(self, amount): self.balance += amount
    def withdraw(self, amount): self.balance -= amount
acc1 = account('1234', 100); acc1.deposit(500)
acc1.withdraw(200); print(acc1.balance)
```

6. 下列 Python 语句的程序运行结果为________。

```
class A:
    def __init__(self, a, b, c): self.x = a + b + c
a = A(6,2,3); b = getattr(a, 'x'); setattr(a, 'x', b+1);print(a.x)
```

7. 阅读下面的 Python 语句，请问输出结果是什么？

```
d1 = {'a':[1,2], 'b':2}; d2 = d1.copy(); d1['a'][0] = 6
sum = d1['a'][0] + d2['a'][0]; print(sum)
```

8. 阅读下面的 Python 语句，请问输出结果是什么？

```
from copy import *
d1 = {'a':[1,2], 'b':2}; d2 = deepcopy(d1); d1['a'][0] = 6
sum = d1['a'][0] + d2['a'][0]; print(sum)
```

9. 下列 Python 语句的程序运行结果为________。

```
list1 = [1,2,3]; list2 = [3,4,5];dict1 = {'1':list1, '2':list2};dict2 = dict1.copy()
dict1['1'][0] = 15; print(dict1['1'][0] + dict2['1'][0])
```

10. 下列 Python 语句的程序运行结果为________。

```
import copy
list1 = [1,2,3]; list2 = [3,4,5]; dict1 = {'1':list1, '2':list2}
dict2 = copy.deepcopy(dict1); dict1['1'][0] = 15
print(dict1['1'][0] + dict2['1'][0])
```

11. 下列 Python 语句的程序运行结果为________。

```
class Person:
    def __init__(self, id): self.id = id
mary = Person(123); mary.__dict__['age'] = 18
mary.__dict__['gender'] = 'female'; print(mary.age + len(mary.__dict__))
```

9.11 上机实践

1. 完成本章中的例 9.1～例 9.53,熟悉 Python 语言面向对象的程序设计。

2. 编写程序,创建类 MyMath,计算圆的周长和面积以及球的表面积和体积,并编写测试代码,结果均保留两位小数。程序运行效果参见图 9-3。

3. 编写程序,创建类 Temperature,其包含成员变量 degree(表示温度)以及实例方法 ToFahrenheit()(将摄氏温度转换为华氏温度)和 ToCelsius()(将华氏温度转换为摄氏温度),并编写测试代码。程序运行效果参见图 9-4。

```
请输入半径: 5
圆的周长 = 31.42
圆的面积 = 78.54
球的表面积 = 314.16
球的体积 = 523.60
```

图 9-3 求圆的周长和面积以及球的表面积和体积的程序的运行效果

```
请输入摄氏温度: 30
摄氏温度 = 30.0, 华氏温度 = 86.0
请输入华氏温度: 86
华氏温度 = 86.0, 摄氏温度 = 30.0
```

图 9-4 摄氏温度和华氏温度相互转换的程序的运行效果

9.12 案例研究:文本相似度比较分析

面向对象的程序设计是建模和解决实际问题的重要方法途径。Python 第三方库大多采用面向对象的程序设计方法。

本案例设计和实现有关文本相似度比较的类 Vector 和 Sketch,以帮助读者进一步提高设计 Python 类来解决实际问题的能力。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

第 10 章

模块和客户端


视频讲解

模块对应于 Python 源代码文件。在 Python 模块中可以定义变量、函数和类。多个功能相似的模块(源文件)可以组成一个包(文件夹)。用户通过导入其他模块,可以使用该模块中定义的变量、函数和类,从而重用其功能。在 Python 中包含了数量众多的模块,可以实现不同的功能和应用。

10.1　模块化程序设计的概念

10.1.1　模块化程序设计

如果程序中包含了多个可以复用的函数或类,则通常把相关的函数和类分组包含在单独的模块(module)中。这些提供计算功能的模块称为模块(或函数模块),导入并使用这些模块的程序则称为客户端程序。

把计算任务分成不同模块的程序设计方法称为模块化编程(modular programming)。使用模块可以将计算任务分解为大小合理的子任务,并实现代码的重用功能。

10.1.2　模块的 API

在客户端使用模块提供的函数时,无须了解其实现细节。模块和客户端之间遵循的契约称为 API(Application Programming Interface,应用程序编程接口)。

API 用于描述模块中提供的函数的功能和调用方法。

模块化程序设计的基本原则是先设计 API(即模块提供的函数或类的功能描述),然后实现 API(即编写程序,实现模块函数或类),最后在客户端中导入并使用这些函数或类。

通过内置函数 help()可以查看 Python 模块的 API。其语法格式为:

```
import 模块名
help(模块名)
```

在查看模块 API 之前,需要使用 import 语句导入模块,也可以使用 Python 在线帮助查看模块的 API。

【例 10.1】　通过内置函数 help()查看 math 模块的 API,过程和部分结果如图 10-1 所示。

【例 10.2】　通过 Python 在线帮助查看 math 模块的 API。

(1) 运行 Python 的内置集成开发环境 IDLE。

(2) 打开 Python Docs。选择 IDLE 中的 Help|Python Docs 命令,打开 Python 帮助文档。

```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.
```

图 10-1　通过内置函数 help()查看 math 模块的 API

（3）定位到 math 模块，查看其 API，如图 10-2 所示。

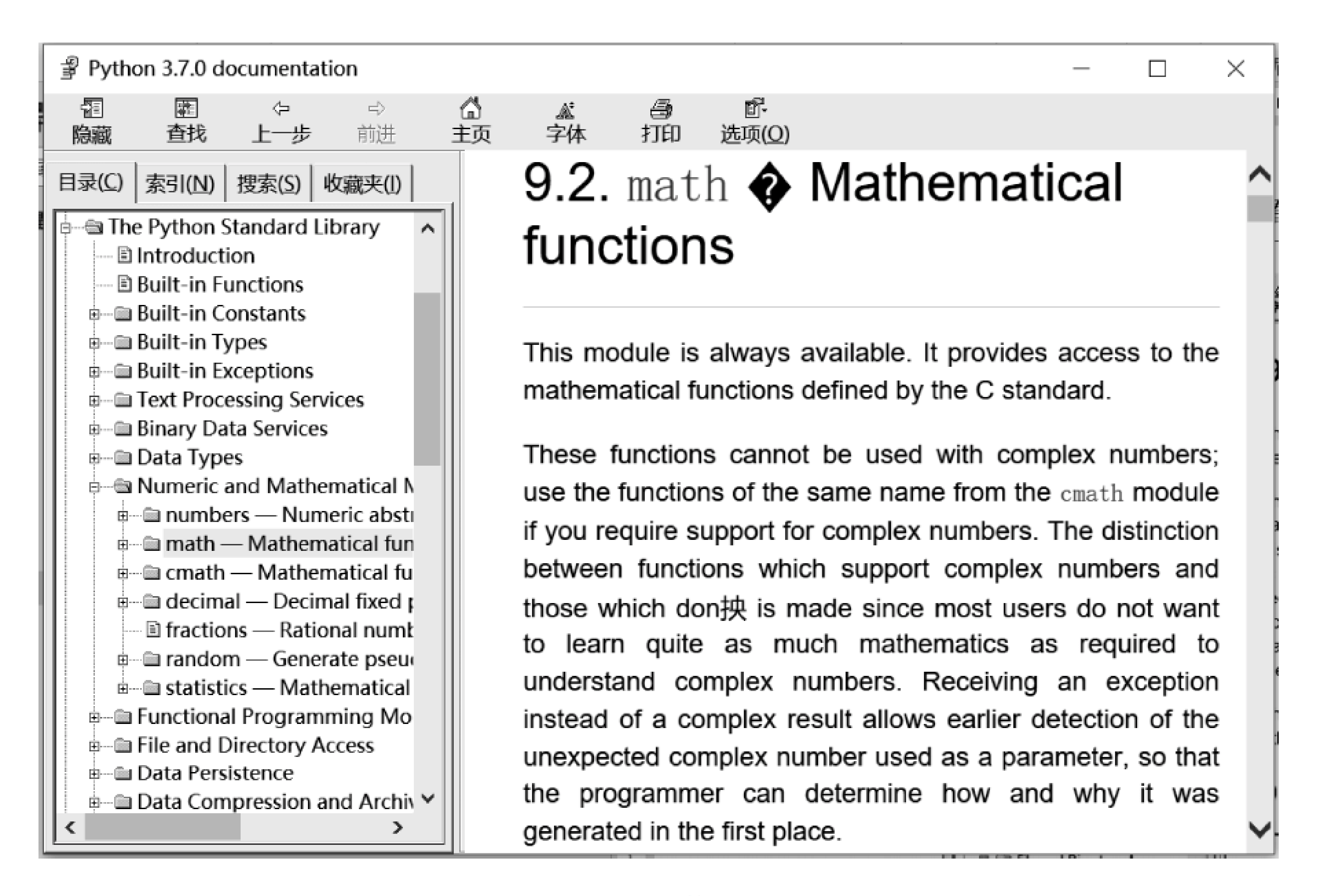


图 10-2　math 模块的 API

10.1.3　模块的实现

“实现”是指实现用于重用的函数或类的代码，模块的实现就是若干实现函数或类的代码的集合，保存在一个扩展名为.py 的文件中。

模块的实现必须遵循 API 规约，可以采用不同算法实现 API，这为模块的改进和版本升级提供了无缝对接，只需要使用遵循 API 的新的实现，所有客户端程序无须修改即可正常运行。

模块通常是使用 Python 语言编写的程序（.py 文件）。本书后续章节将详细阐述。

注意：Python 内置模块使用 C 编写并已链接到 Python 解释器内，还可以使用 C 或 C++扩展编写模块（编译为共享库或 DLL 文件）。

10.1.4　模块的客户端

客户端遵循 API 提供的调用接口，导入和调用模块中实现的函数功能。

API 允许任何客户端直接使用模块，而无须检测模块中定义的代码，例如可以直接使用模块 math 和 random。

【例 10.3】 模块的客户端示例(client.py)。在[0,π]区间均匀输出函数 $y=\sin(x)+\sin(5x)$ 对应的 n 个函数值。其中，n 由命令行的第一个参数所确定。

```
import math
import sys
n = int(sys.argv[1])
for i in range(n+1):
    x = math.pi * i / n
    y = math.sin(x) + math.sin(5 * x)
    print(x, y)
```

程序运行结果如图 10-3 所示。

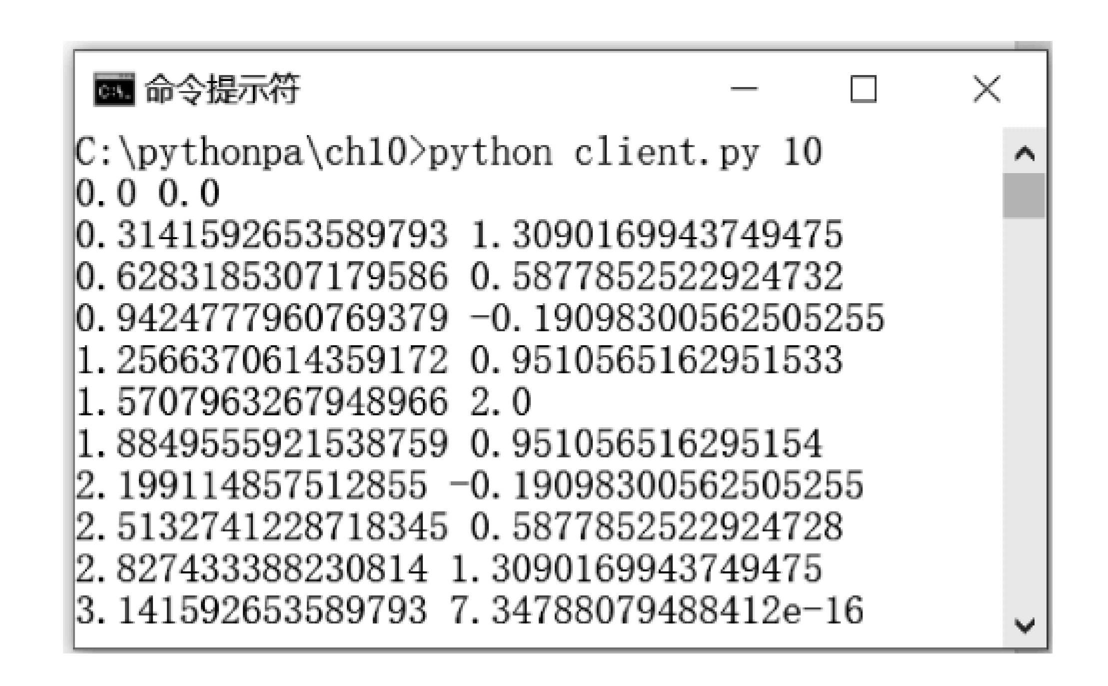


图 10-3　模块的客户端示例程序运行结果

10.1.5　模块化程序设计的优越性

模块化程序设计是现代程序设计的基本理念之一，具有如下优越性。

(1) 可以编写大规模的系统程序：通过把复杂的任务分解为子任务可以实现团队合作开发，完成大规模的系统程序。

(2) 控制程序的复杂度：分解后的子任务的实现模块代码规模一般控制在数百行之内，从而可以控制程序的复杂度，各代码调试可以限制在少量的代码范围。

(3) 实现代码重用：一旦实现了通用模块(如 math、random 等)，任何客户端都可以通过导入模块直接重用代码，而无须重复实现。

(4) 增强可维护性：模块化程序设计可以增强程序的可维护性。通过改进一个模块的实现可以使得使用该模块的客户端同时被改进。

10.2　模块的设计和实现

10.2.1　模块设计的一般原则

模块设计的指导性原则一般包括如下几点。

(1) 先设计 API，再实现模块。

(2) 控制模块的规模,只为客户端提供需要的函数。实现包含大量函数的模块会导致模块的复杂性。例如,在 Python 的 math 模块中就不包含正割函数、余割函数和余切函数,因为这些函数很容易通过函数 math.sin()、math.cos()和 math.tan()的计算而得。

(3) 在模块中编写测试代码,并消除全局代码。

(4) 使用私有函数实现不被外部客户端调用的模块函数。

(5) 通过文档提供模块帮助信息。

10.2.2 API 设计

API 定义客户端和实现之间的契约。API 是一个明确的规范,规定"实现"的具体功能是什么。

API 通常由两部分组成,即可用函数的签名的精确规范,以及描述函数作用的非正式自然语言描述。API 一般使用表格的形式描述模块中的变量、函数和类。

在编写一个新模块时,建议先设计 API,然后实现模块。

【例 10.4】 设计实现算术四则运算的模块(my_math1.py)的 API。设计结果如表 10-1 所示。

表 10-1 my_math1.py 模块的 API

函数调用	功能描述
add(x,y)	加法函数 add(x,y)
sub(x,y)	减法函数 sub(x,y)
mul(x,y)	乘法函数 mul(x,y)
div(x,y)	除法函数 div(x,y)

10.2.3 创建模块

Python 模块对应于包含 Python 代码的源文件(其扩展名为.py),在文件中可以定义变量、函数和类。

在模块中除了可以定义变量、函数和类之外,还可以包含一般的语句,称之为主块(全局语句)。当运行该模块或者导入该模块时,主块语句将依次执行。

一般而言,独立运行的源代码中主要包含主块,以实现相应的功能。作为库的模块,主要包含可供调用的变量、函数和类,还可以包含用于测试的主块代码。

值得注意的是,主块代码语句只在模块第一次被导入时执行,重复导入时不会多次导入多次执行。

【例 10.5】 创建模块 my_math1.py,在模块中定义算术四则运算。

```
PI = 3.14                         #定义常量
def add(x, y):                    #定义函数
    return x + y                  #加
def sub(x, y):                    #定义函数
    return x - y                  #减
def mul(x, y):                    #定义函数
    return x * y                  #乘
def div(x, y):                    #定义函数
    return x / y                  #除
```

10.2.4 模块的私有函数

在实现模块时,有时候需要在模块中定义仅在模块中使用的辅助函数。辅助函数不提供给客户端直接调用,故称之为私有函数。

按惯例,Python 程序员使用以下画线开始的函数名作为私有函数。私有函数在客户端不应该直接调用,故 API 中不包括私有函数。Python 语言没有强制不允许调用私有函数的机制,程序员应该避免直接调用私有函数。

【例 10.6】 创建模块 normal.py,实现正态分布的概率密度函数 PDF,其函数形式为 $f(x|\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}}$。

```
import math
def _phi(x):
    return math.exp( - x * x/2.0) / math.sqrt(2 * math.pi)
def pdf(x, mu = 0.0, sigma = 1.0):
    return _phi(float((x - mu) / sigma)) / sigma
#测试代码
if __name__ == '__main__': #如果独立运行,则运行测试代码
    for i in range(0,101):
    print(i, pdf(i, mu = 78, sigma = 10))
```

程序运行结果如下。

```
0 2.4528552856964323e-15
1 5.324148372252943e-15
...
```

这也就是期望值为 78、标准差为 10 时各分数的概率。

10.2.5 模块的测试代码

每个模块都有一个名称,通过特殊变量__name__可以获取模块的名称。例如:

```
>>> import os
>>> os.chdir(r'c:\pythonpa\ch10')
>>> import my_math1
>>> my_math1.__name__                    #输出:'my_math1'
```

特别地,当一个模块被用户单独运行时,其__name__的值为'__main__'。故可以把模块源代码文件的测试代码写在相应的测试判断中,以保证只有单独运行时才会运行测试代码。

【例 10.7】 创建模块 my_math2.py(测试代码只有独立运行时才执行)。

```
PI = 3.14                                #定义常量
def add(x, y):                           #定义函数
    return x + y                         #加
def sub(x, y):                           #定义函数
    return x - y                         #减
def mul(x, y):                           #定义函数
    return x * y                         #乘
def div(x, y):                           #定义函数
    return x / y                         #除
#测试代码
def main():
    print('123 + 456 = ', add(123, 456)) #加
```

```
    print('123 - 456 = ', sub(123, 456))    #减
    print('123 * 456 = ', mul(123, 456))    #乘
    print('123 / 456 = ', div(123, 456))    #除
if __name__ == '__main__':                   #如果独立运行,则运行测试代码
    main()
```

程序运行结果如下。

```
123 + 456 = 579
123 - 456 = -333
123 * 456 = 56088
123 / 456 = 0.26973684210526316
```

10.2.6 编写模块文档字符串

在程序源代码中,可以在特定的地方添加描述性文字,以说明包、模块、函数、类、类方法的相关信息。

在函数的第一个逻辑行的字符串称为函数的文档字符串。函数的文档字符串用于提供有关函数的帮助信息。

文档字符串一般遵循下列惯例:文档字符串是一个多行字符串;首行以大写字母开始,以句号结尾;第二行是空行;从第三行开始是详细的描述。

用户可以使用 3 种方法抽取函数的文档字符串帮助信息:①使用内置函数 help(函数名);②使用函数的特殊属性函数名.__doc__;③第三方自动化工具也可以抽取文档字符串信息,以形成帮助文档。

【例 10.8】 查看文档字符串示例帮助信息。

```
>>> help(abs)
    Help on built-in function abs in module builtins:
    abs(x, /)
        Return the absolute value of the argument.
>>> print(abs.__doc__)                  #输出:Return the absolute value of the argument.
```

同样,在包的__init__.py 中注释,成为包的文档字符串;在文件头部注释,成为模块的文档字符串;在 class 声明后第一个逻辑行注释,成为类文档字符串。

【例 10.9】 文档字符串示例(doc.py)。

```
"""doc 模块说明文档"""                     #模块注释
def d2b(i):                                #定义类 d2b
    """函数 d2b()的说明文档"""
    print(bin(i))
class Doc:                                 #定义类 Doc
    """类 Doc 的说明文档"""
    def sayHello(self):                    #定义类 Doc 的方法 sayHello()
        """方法 sayHello()的说明文档"""
        print('hi')
```

运行过程和结果如下。

```
>>> import doc
>>> doc.__doc__                      #输出:'doc 模块说明文档'
>>> doc.d2b.__doc__                  #输出:'函数 d2b()的说明文档'
>>> doc.Doc.__doc__                  #输出:'类 Doc 的说明文档'
>>> doc.Doc.sayHello.__doc__         #输出:'方法 sayHello()的说明文档'
```

10.2.7　按字节编译的.pyc 文件

在导入模块时，Python 解释器为加快程序的启动速度，会在与模块文件同一目录的 __pycache__子目录下生成.pyc 文件。

.pyc 文件是经过编译后的字节码，这样下次导入时如果模块源代码.py 文件没有修改（通过比较两者的时间戳），则直接导入.pyc 文件，从而提高程序效率。

按字节编译的.pyc 文件是在导入模块时由 Python 解释器自动完成，无须程序员手动编译。

10.3　模块的导入和使用

Python 中包含了数量众多的模块，通过 import 语句和 reload()函数，可以导入模块，并使用其定义的功能。

10.3.1　导入模块和使用模块

使用 import 语句可以导入模块。其基本形式如下：

```
import 模块名                          #导入模块
import 模块 1, 模块 2, …,模块 n        #导入多个模块
import 模块名 as 模块别名              #导入模块并使用别名
```

其中，模块名是要导入的模块的名称。注意，模块名区分大小写。

一般在 Python 源程序的开始位置导入其他模块。在导入模块后，可以使用全限定名称访问模块中定义的成员，即：

```
模块名.函数名/变量名                   #使用包含模块的全限定名称调用模块中的成员
```

【例 10.10】 导入模块并使用模块函数示例。

```
>>> import math
>>> math.pi                            #输出:3.141592653589793
>>> math.trunc(1.23)                   #输出:1
>>> import os, sys
>>> os.getcwd()                        #输出:'C:\\Pythonpa\\ch10'
>>> import os as operatingSystem
>>> operatingSystem.getcwd()           #输出:'C:\\Pythonpa\\ch10'
```

10.3.2　导入模块中的成员

Python 使用 from … import 语句直接导入模块中的成员。其基本形式如下：

```
from 模块名 import 成员名              #导入模块中的具体成员
成员名                                 #直接调用
```

如果希望同时导入一个模块中的多个成员，可以采用下列形式：

```
from 模块名 import 成员名 1, 成员名 2, … , 成员名 n
```

如果希望同时导入一个模块中的所有成员，则可以采用下列形式：

```
from 模块名 import *
```

【例 10.11】 导入模块中的成员示例。

```
>>> from math import pi, sin
>>> sin(pi/2)                               #输出:1.0
>>> from os import *
>>> getcwd()                                #输出:'C:\\Pythonpa\\ch10'
```

注意：虽然 from…import 语句可以简化代码，但读者应避免使用，因为这样可能导致名称冲突（例如导入多个模块时，多个模块中可能存在同一个名称的函数），且导致程序的可读性差（例如导入多个模块时，无法准确确定某个名称的函数具体属于哪一个模块）。

10.3.3 重新加载模块

importlib 模块中的 reload() 函数用于重新加载之前导入过的模块。一般用于在交互式执行 Python 代码不退出解释器的情况下，重新加载已更改的 Python 模块。

注意：重新加载内存中不存在的模块（未导入过）会导致运行时错误。

【例 10.12】 重新加载模块示例。

```
>>> from importlib import reload
>>> reload(os)                              #报错.NameError: name 'os' is not defined
>>> import os
>>> reload(os)
<module 'os' from 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\lib\\os.py'>
```

10.3.4 动态导入模块

使用内置函数__import__()可以动态导入模块：

```
_m = __import__(name)                       #导入模块 name 到_m
```

内置函数__import__()具有更大的灵活性，例如要导入的模块 name 可以是计算的结果字符串，但一般不直接使用。事实上，import 语句在内部调用该函数。

【例 10.13】 动态导入模块示例。

```
>>> s = 'os' + '.' + 'path'
>>> _m = __import__(s)
>>> _m.curdir                               #输出:'.'
```

10.4 包

10.4.1 包的概念

在大型项目中往往需要创建许多模块，这些功能相似的模块可以使用包组成层次组织结构，以便于维护和使用。

Python 模块是.py 文件，而包是文件夹。通常，只要文件夹中包含一个特殊的文件__init__.py，则 Python 解释器就将该文件夹作为包，其中的模块文件(.py 文件)属于包中的模块。

特殊文件__init__.py 可以为空，也可以包含属于包的代码，当导入包或该包中的模块时执行__init__.py。

包可以包含子包，没有层次限制。使用包可以有效避免名称空间冲突。

【例 10.14】 包示例，如图 10-4 所示。

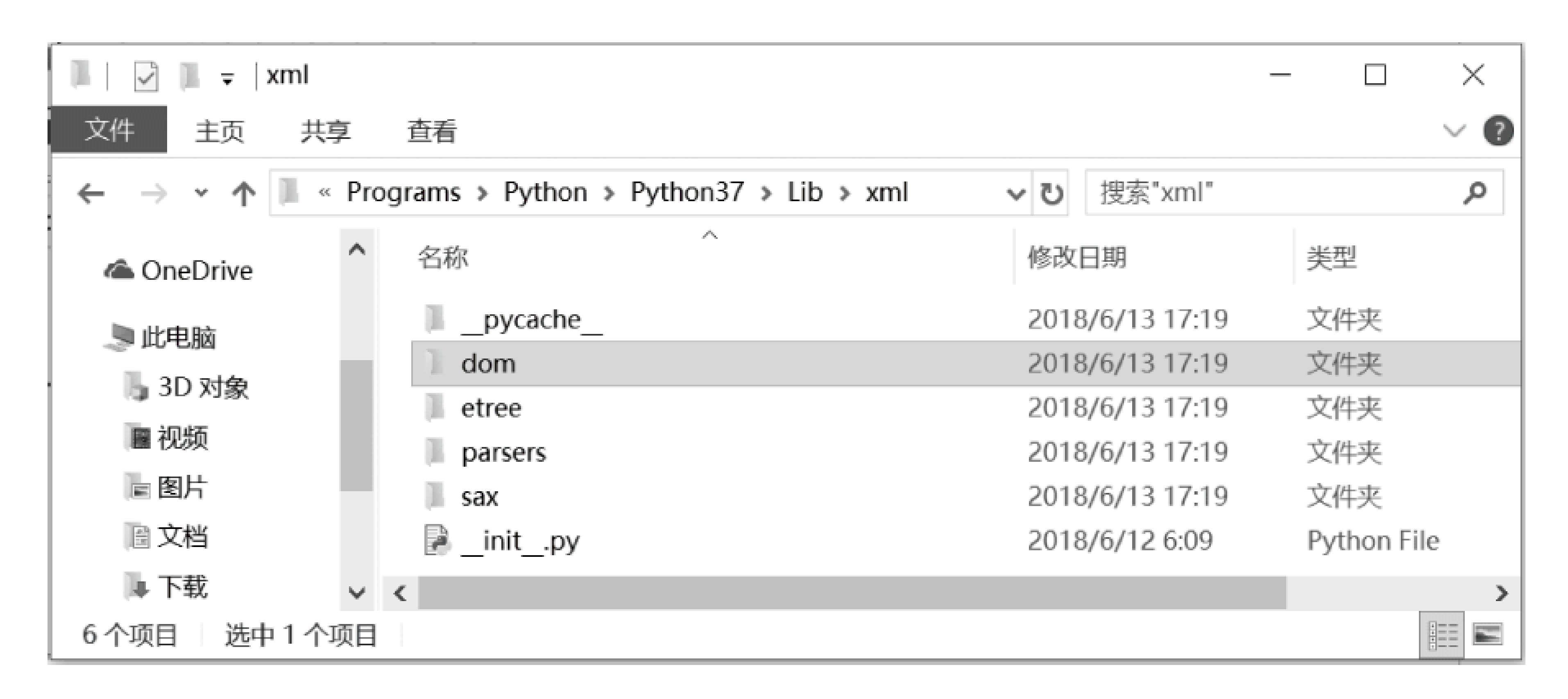


图 10-4　包示例

图 10-4 所示的包示例目录结构表明，在 Python 标准库中(Lib 目录下)包含包 xml。xml 是顶级包，包含子包 dom、etree、parsers 和 sax。

10.4.2　创建包

包和模块组成的层次组织结构对应于文件夹和模块文件。

创建包，首先需要在指定目录中创建对应包名的目录，然后在该目录下创建一个特殊文件 __init__.py，最后在该目录下创建模块文件。

【例 10.15】　创建包示例。在“C:\pythonpa\ch10\”目录中创建如下目录结构：

```
\package1
    __init__.py
    \subPackage1
        __init__.py
        module11.py
        module12.py
        module13.py
    \subPackage2
        __init__.py
        module21.py
        module22.py
```

其中，package1 是顶级包，包含子包 subPackage1 和 subPackage2，而包 subPackage1 包含模块 module11、module12 和 module13，包 subPackage2 包含模块 module21 和 module22。

10.4.3　包的导入和使用

在使用 import 语句导入包中的模块时需要指定对应的包名。其基本形式如下：

```
import [包名 1.[包名 2.…]].模块名            #导入包中模块
```

其中，包名是模块的上层组织包的名称。注意，包名和模块名区分大小写。

在导入包中模块后，可以使用全限定名称访问包中模块定义的成员：

```
[包名 1.[包名 2.…]].模块名.函数名            #使用全限定名称调用模块中的成员
```

用户也可以使用 from … import 语句直接导入包中模块的成员。其基本形式如下：

```
from [包名 1.[包名 2.…]].模块名 import 成员名    #导入模块中的具体成员
```

同一个包/子包的模块，可以直接导入相同包/子包的模块，而不需要指定包名。这是因为同一个包/子包的模块位于同一个目录。例如，如果包 subPackage2 中包含模块 module21 和 module22，则在模块 module22 中可以通过 import module21 直接导入 module21。

当直接导入包时（import 包名），将执行包目录下的__init__. py（其中创建的名称有效），但不会导入目录下的模块和子包（子目录）。例如：

```
>>> import xml
>>> xml.dom #报错.AttributeError: module 'xml' has no attribute 'dom'
```

使用“from 包名 import ＊”并不会自动导入一个包中的所有模块（子包），而是导入__init__. py 中指定的模块或子包。例如：

```
>>> from xml import *
>>> xml.dom
<module 'xml.dom' from 'C:\\Program Files\\Python37\\lib\\xml\\dom\\__init__.py'>
```

【例 10.16】 包的导入和使用示例。

```
>>> from xml.dom import minidom
>>> doc = minidom.Document()
```

10.5 模块的导入顺序

10.5.1 导入模块时的搜索顺序

在导入模块时，解释器按下列目录搜索路径和文件搜索顺序查找并导入文件。目录搜索路径如下。

（1）当前目录。启动交互式 Python 的目录，或者 Python 主程序位于的目录。

（2）操作系统环境变量 PYTHONPATH 中指定的目录。

（3）Python 标准库目录。

各目录下的文件（目录也是文件的一种）查找顺序依次如下（以 import foo 为例）。

（1）包：定义为一个包的目录 foo。

（2）扩展模块：foo. so、foomodule. so、foomodule. sl 或 foomodule. dll（已编译扩展）。

（3）优化模块：foo. pyo（仅在使用-O 或-OO 选项时）。

（4）编译模块：foo. pyc。

（5）Python 模块：foo. py。

说明：

（1）当一个模块（. py）第一次被导入时，Python 解释器会自动将其编译为字节码格式（. pyc）。后续的导入操作直接读取. pyc 文件（如果. py 文件被修改，则会重新生成. pyc 文件）。

（2）当 Python 解释器使用-O 选项时，将生成优化代码（. pyo）。优化代码去掉断言及其他调试信息，使得代码体积更小、速度更快。如果 Python 解释器使用-OO 选项代替-O 选项，则文档字符串也会被忽略。

（3）如果在 sys. path 提供的所有路径均查找失败，则解释器会继续在内建模块中查找；如果再次查找失败，则抛出 ImportError 异常。

（4）使用 import 语句搜索文件时，文件名是大小写敏感的。

10.5.2 模块搜索路径

sys 模块的 sys.path 属性返回一个路径列表。在使用 import 语句导入模块时,系统将自动从该列表的路径中搜索模块,如果没有找到,则程序报错。

【例 10.17】 模块搜索路径示例。

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip', 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\DLLs', 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\lib', 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37', 'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages']
```

其中,第一个''表示当前目录;最后一个'C:\\Users\\jh\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages'用于扩展模块。建议用户自定义模块放置在这两个位置。

在程序中也直接修改 sys.path 列表,以添加模块搜索路径。但这种修改是临时的,即只适用于包含该代码的程序。

【例 10.18】 临时增加模块搜索路径示例。

```
>>> import sys
>>> sys.path.append('C:\\pythonpa\works')
```

10.5.3 dir()函数

在模块中定义的成员,包括变量、函数和类等,可以通过内置的函数 dir()查询,也可以通过 help()函数查询其帮助信息。dir()函数的基本形式如下:

```
• dir()              #不带参数,列举当前模块的所有成员
• dir(模块名)        #列举指定模块的所有成员
• dir(类/对象)       #列举指定类的所有成员.注意,Python 中所有的成员都是对象
```

在列举的成员中包含系统定义的特殊意义的成员(__xxx__形式)。

【例 10.19】 列举模块成员示例。

```
>>> dir()              #列举当前模块的所有成员
['DirEntry', 'F_OK', 'Fib', 'O_APPEND', 'O_BINARY', … ]
>>> a = 10             #增加变量 a
>>> dir()              #列举当前模块的所有成员,包含 a
>>> del a              #删除变量 a
>>> dir()              #列举当前模块的所有成员,不包含 a
>>> import math
>>> dir(math)          #列举 math 模块的所有成员
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> dir(10)            #列举 10(int 对象)的所有成员
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne_
```

```
_', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__
reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror
__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '_
_setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor
__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>> dir(str)                  #列举类的所有成员
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__
', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '
__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__
', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', '
expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit
', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', '
rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

10.6 名称空间与名称查找顺序

10.6.1 名称空间概述

在 Python 程序中，每一个名称（变量名、函数名或类型名）都有一个作用范围。在其作用范围之内，可以直接引用该名称；在其作用范围之外，该名称不存在，引用该名称将导致 NameError 错误。这个作用范围称为名称空间，也称为命名空间。

10.6.2 名称查找顺序

当代码中使用名称 x 时，Python 解释器把 x 解释为对象名（对象、函数、变量等），并按如下名称空间顺序查找以 x 命名的对象。

（1）局部名称空间：当前函数或类的方法中定义的局部变量。

（2）全局名称空间：当前模块（.py 文件）中定义的变量、函数或类。

（3）内置名称空间：对每个模块都是全局的。作为最后的尝试，Python 将假设 x 是内置函数或变量。

如果最后查找不到以 x 命名的对象，则抛出 NameError 错误。

【例 10.20】 名称查找示例。

```
>>> math.e                    #报错.NameError: name 'math' is not defined
>>> import math
>>> math.e                    #输出:2.718281828459045
```

在导入 math 模块前，Python 查找不到以 e 命名的对象，故抛出错误 NameError。在导入 math 模块之后，查找到 math 模块中的全局变量 e，故返回其值。

10.6.3 顶层模块和__name__变量

Python 程序通常由多个模块组成，其中一个模块作为包含应用程序的启动代码，这个模块称为顶层模块。其余模块本质上是“库”模块，由顶级模块导入，包含应用程序使用的函数和类。

Python 使用特殊变量__name__来标记一个模块是否为顶层模块。

（1）如果模块是作为一个正在运行的顶层模块，则其属性__name__设置为字符串__

main__。

(2) 如果模块被另一个模块(不管是顶层模块或其他模块)导入,则其属性__name__设置为模块的名称。

因此,在模块中编写仅当作为顶层模块运行时才执行的代码可以使用下列语句:

```
if __name__ == "__main__":
    #作为顶层模块运行时要执行的代码
```

【例 10.21】 顶层模块和__name__变量示例 1(lib_module.py)。

```
print("lib_module.py:__name__ = {}".format(__name__))
if __name__ == "__main__":
    print("lib_module.py 作为主模块运行时执行的代码")
```

程序运行结果如下。

```
lib_module.py:__name__ = __main__
lib_module.py 作为主模块运行时执行的代码
```

【例 10.22】 顶层模块和__name__变量示例 2(top_module.py)。

```
import lib_module
print("top_module.py:__name__ = {}".format(__name__))
if __name__ == "__main__":
    print("top_module.py 作为主模块运行时执行的代码")
```

程序运行结果如下。

```
lib_module.py:__name__ = lib_module
top_module.py:__name__ = __main__
top_module.py 作为主模块运行时执行的代码
```

10.6.4 Python 解释器

在使用 Python 解释器交互式执行 Python 代码时,Python 解释器是顶层模块,其中定义的名称是全局变量,属于全局名称空间。

【例 10.23】 Python 解释器示例。

```
>>> __name__          #输出:'__main__'
```

10.6.5 全局名称空间

在解释器命令行或在模块中的函数之外赋值定义的名称,其作用范围是与命令行或者整个模块关联的名称空间,称之为全局作用范围、全局名称空间。

在全局作用范围中定义或者导入的对象名称(变量)被称为全局名称(变量)。

【例 10.24】 使用 dir()查看 Python 解释器中的全局名称。

```
>>> dir()                 #查看全局名称空间:默认创建和导入的名称
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__'
]
>>> a = 1                 #定义对象变量名称
>>> import math           #导入模块名称 math
>>> dir()                 #再次查看全局名称空间,增加了名称 a 和 math
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__','__spec
__', 'a', 'math']
>>> dir(__builtins__)     #查看内置模块的名称空间
```

```
['ArithmeticError', …, 'zip']
>>> dir(math)               # 查看 math 模块的名称空间
['__doc__', …, 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

10.6.6 局部名称空间

在一个函数的函数体（或类的方法的方法体）中定义的名称，其作用范围为函数体（方法体）局部，称之为局部作用范围、局部名称空间。其名称空间是与函数调用关联的名称空间。

在执行函数调用时分配的名称被称为本地名称，它们是函数调用中的局部名称，这就是函数调用栈。函数的局部名称只存在于与函数调用相关的名称空间中，它们具有如下特点。

（1）仅对函数中的代码可见。

（2）不会影响函数以外定义的名称，即使它们的名称相同。

（3）仅在函数执行期间存在；在函数开始执行之前不存在，并且在函数完成执行后不再存在。

【例 10.25】 递归调用栈和局部名称空间示例（recursive_stack.py）。

```
def vertical(n):
    """依次垂直输出整数的各数字"""
    if n < 10:                  # 基本情况:n 为一位数时直接输出
        print(n)
    else:                       # 递归情况:当为多位数时,整除 10 后递归调用
        vertical(n//10)
        print(n % 10)           # 输出余数
if __name__ == "__main__":
    vertical(687)
```

（1）打开编辑源代码。使用 IDLE 打开“C:\pythonpa”下的 recursive_stack.py。

（2）设置断点。右击第三行代码，通过快捷菜单命令 Set Breakpoint 设置断点。

（3）打开调试器。在 Python 解释器命令行窗口中通过菜单命令 Debug|Debugger，打开调试器控制窗口。

（4）调试运行程序。在 IDLE 窗口中按 F5 键，程序开始运行。

（5）单步执行并查看调用堆栈。在 Debug Control 窗口中通过单击 Over 按钮单步执行语句，并查看调用堆栈（即局部变量），如图 10-5 所示。

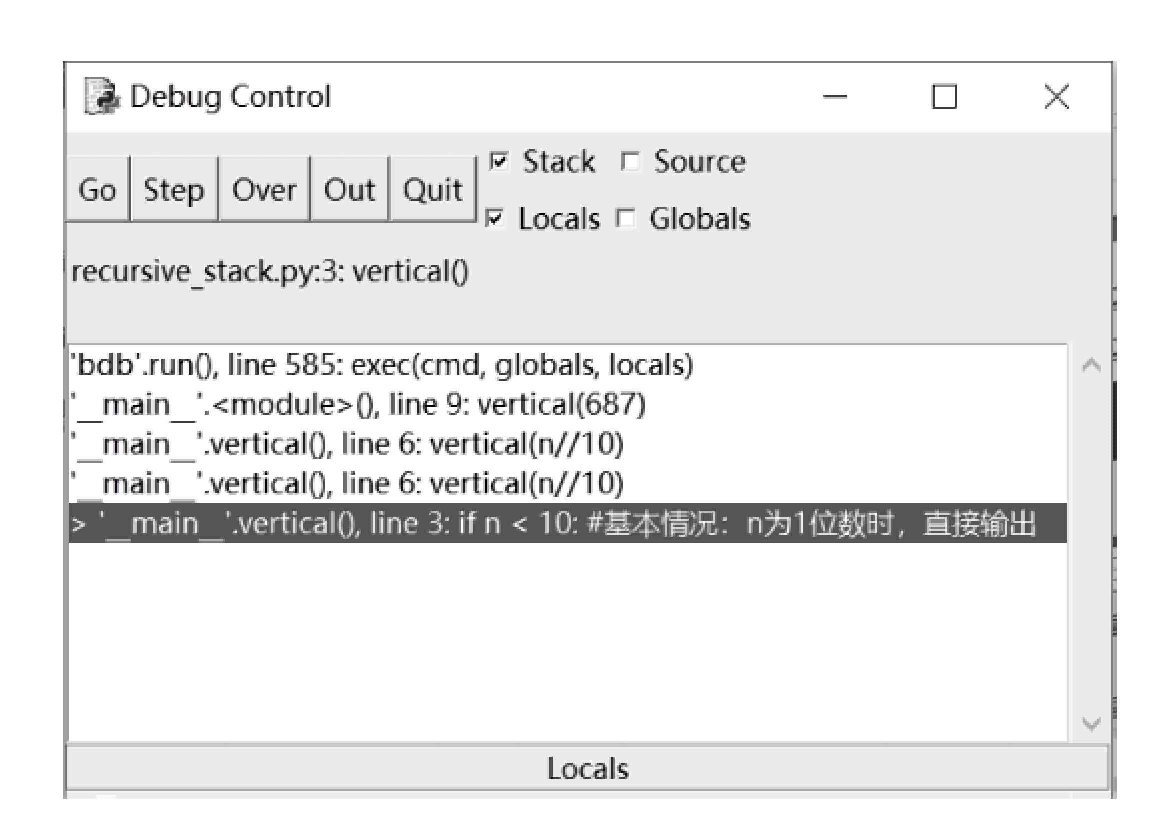


图 10-5 单步执行并查看调用堆栈

10.6.7　类和对象名称空间

在 Python 中，类和对象也都关联一个名称空间。类名称空间的名称是类的名称，存储在名称空间中的名称是类的属性和类方法(包括继承于父类名称和自身定义的名称)。例如，列表是名为 list 的名称空间，其中包含列表类的方法和运算符的名称。类的对象实例也是一个名称空间，其中的名称包括其所属类的属性以及对象本身的属性。

【例 10.26】　类和对象名称空间示例。

```
>>> list                        #输出:<class 'list'>
>>> dir(list)
['__add__', …, 'pop', 'remove', 'reverse', 'sort']
>>> a = []
>>> a.append(1)
>>> dir(a)
['__add__', …, 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

10.7　复　习　题

一、填空题

1. 在 Python 中包含了数量众多的模块，通过________语句可以导入模块，并使用其定义的功能。

2. 在 Python 中假设有模块 m，如果希望同时导入 m 中的所有成员，则可以采用________的导入形式。

3. 在 Python 中使用内置函数________也可以导入模块。

4. Python 中 sys 模块的________属性可以返回一个路径列表。

5. Python 中的每个模块都有一个名称，通过特殊变量________可以获取模块的名称。特别地，当一个模块被用户单独运行时，模块名称为________。

6. 在 Python 模块中定义的所有成员，包括变量、函数和类等，可以通过内置函数________查询，也可以通过________函数查询其帮助信息。

二、思考题

1. 什么是模块？模块是如何导入解释器的？分别有哪几种方法？
2. 在 Python 中包和模块是什么关系？包和模块组成的层次组织结构分别对应于什么？
3. 在 Python 中创建包的基本步骤和内容是什么？如何导入和使用包？
4. 在 Python 中导入模块时一般采用什么搜索顺序？
5. 在 Python 中名称空间与名称查找顺序是什么？

10.8　上 机 实 践

1. 完成本章中的例 10.1～例 10.26，熟悉 Python 语言模块和客户端程序设计。

2. 编写程序，创建一个实现＋、－、*、/和 ** (幂)运算的模块 MyMath.py，并编写测试代码。其运行效果参见图 10-6。

3. 编写程序，创建一个求圆的面积和球体体积的模块 AreaVolume. py，并编写只有独立运行时才执行的测试代码，要求输入半径，输出结果保留两位小数。其运行效果参见图 10-7。

```
123 + 100 = 223
123 - 100 = 23
123 * 100 = 12300
123 / 100 = 1.23
2 ** 10 = 1024
```

图 10-6 实现运算程序的运行效果

```
请输入半径：5.1
圆面积=81.67
球体体积=555.37
```

图 10-7 求面积和体积程序的运行效果

4. 编写程序，创建输出命令行参数个数以及各参数内容的模块 SysArgvs. py，并编写测试代码。其运行效果参见图 10-8。

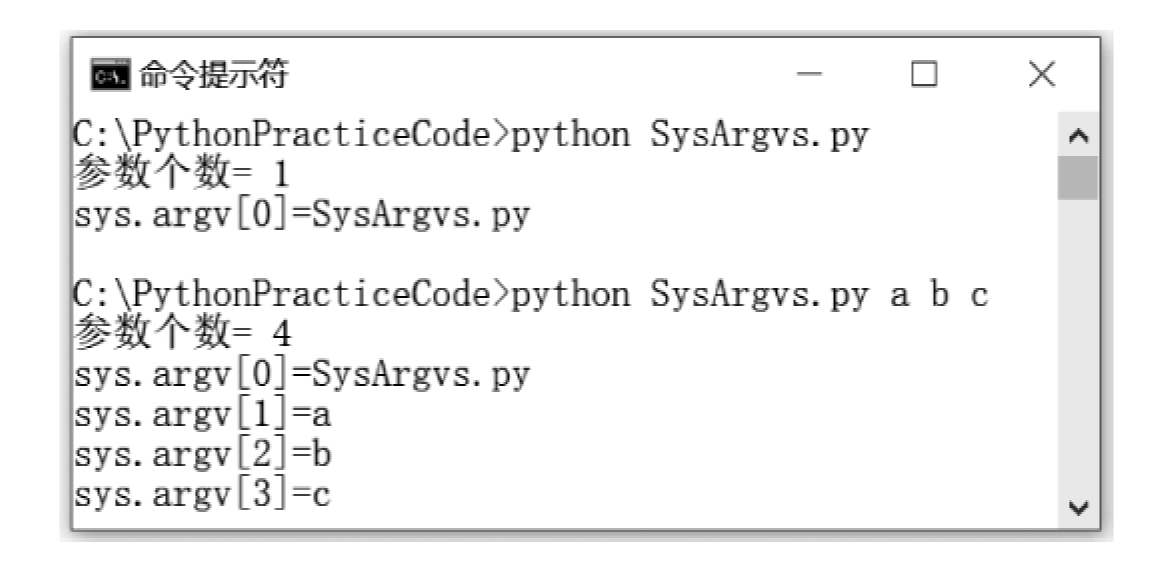


图 10-8 输出命令行参数个数及内容的程序的运行效果

10.9 案例研究：基于模块的库存管理系统

本章案例研究通过一个多模块的库存管理系统案例帮助读者深入了解基于模块的 Python 应用程序的开发流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 11 章

算法与数据结构基础

著名的计算机科学家尼克劳斯·沃思(Niklaus Wirth)指出“程序＝数据结构＋算法”。数据结构用于描述数据，算法则基于数据结构操作数据。Python 的标准库模块提供了若干对象和函数，用于实现各种通用数据结构和算法。



视频讲解

11.1 算法及其性能分析

11.1.1 算法概述

算法是指解决问题的一种方法或一个过程。算法通常使用计算机程序来实现。算法接收待处理的输入数据，然后执行相应的处理过程，最后输出处理的结果。其示意图如图 11-1 所示。



图 11-1　算法结构的示意图

算法的实现为若干指令的有穷序列，具有如下性质。

(1) 输入数据：算法可以接收用于处理的外部数据。

(2) 输出结果：算法可以产生输出结果。

(3) 确定性：算法的组成指令必须是准确、无歧义。

(4) 有限性：算法指令的执行次数必须是有限的，执行的时间也必须是有限的。

在计算机上执行一个算法会产生内存开销和时间开销。算法的性能分析包括以下两个方面：

(1) 时间性能分析。

(2) 空间性能分析。

11.1.2 算法的时间复杂度分析

衡量算法有效性的一个指标是运行时间。算法的运行时间长度与算法本身的设计和所求解问题的规模有关。算法的时间性能分析又称为算法的时间复杂度(Time Complexity)分析。

问题的规模(Size)即算法求解问题的输入量，通常用一个整数表示。例如，矩阵乘积问题的规模是矩阵的阶数，图论问题的规模则是图中的顶点数或边数。

对于问题规模较大的数据，如果算法的时间复杂度呈指数分布，完成算法的时间可能趋向

于无穷大，即无法完成。

一个算法运行的总时间取决于以下两个主要因素。

(1) 每条语句的执行时间成本。

(2) 每条语句的执行次数(频度)。

即一个算法所耗费的时间等于算法中每条语句的执行时间之和。每条语句的执行时间为该语句的执行次数(频度)×该语句执行一次所需的时间。

每条语句执行一次所需的时间取决于实际运行程序的机器的性能。当独立于机器系统分析算法的时间性能时，可以假设每条语句执行一次所需的时间均是单位时间，故一个算法的运行时间等于算法中所有语句的频度之和。

【例 11.1】 算法中语句的频度之和示例(frequency.py)。

```
total = 0
for i in range(n):
    for j in range(n):
      total += a[i][j]
print(total)
```

在例 11.1 中，循环语句运行了 n×n 次，总算法执行语句频度为 n^2+2。

11.1.3 增长量级

对于问题规模 n，假如算法 A 中所有语句的频度之和为 100n+1，算法 B 中所有语句的频度之和为 n^2+n+1，则算法 A 和 B 对于不同问题规模的运行时间对照表如表 11-1 所示。

表 11-1 算法 A 和 B 对于不同问题规模的运行时间对照表

问题规模 n	算法 A 的运行时间	算法 B 的运行时间
10	1001	111
100	10001	10101
1000	100001	1001001
10000	1000001	100010001

由表 11-1 可以看出，随着问题规模 n 的增长，算法的运行时间主要取决于最高指数项。在算法分析中，通常使用增长量级来描述。

增长量级用于描述函数的渐进增长行为，一般使用大 O 符号表示。例如，2n、100n 与 n+1 属于相同的增长量级，记为 O(n)，表示函数随 n 线性增长。

算法分析中常用的增长量级如表 11-2 所示。

表 11-2 常用的增长量级

函数类型	增长量级	举例	说明
常量型	1	`count-= 1`	语句(整数递减)

对数型	$\log_2 N$	`while n > 0:`
		`    n = n // 2`
		`    count += 1`
			除半(二分查找法等)

线性型	n	`for i in range(n):`
		`    if i % 2 != 0: #奇数`
		`        sum_odd += i`
			单循环
			(统计奇数的个数、顺序查找法等)

续表

函数类型	增长量级	举　　例	说　　明
线性对数型	$N\log_2 N$	请参见 11.3.4 归并排序法	分而治之算法(归并排序法等)

二次型	n^2	`for i in range(1, n):`
		`    s = ""`
		`    for j in range(1, n):`
		`        s += str.format("{0:1} * {1:1} = {2:2}", i, j, i * j)`
		`    print(s)`
			两重嵌套循环(打印九九乘法表、冒泡排序算法、选择排序算法、插入排序算法等)

三次型	n^3	`for i in range(n):`
		`  for j in range(i+1, n):`
		`    for k in range(j+1, n):`
		`      if (a[i] + a[j] + a[k]) == 0:`
		`        count += 1`
			三重嵌套循环

11.1.4　算法的空间复杂度分析

衡量算法有效性的另一个指标是内存消耗。对于复杂的算法,如果其消耗的内存超过运行该算法的计算机的可用物理内存,则算法无法正常执行。算法的内存消耗分析又称为算法的空间复杂度(Space Complexity)分析。

Python 语言面向对象特性的主要代价之一是内存消耗。Python 的内存消耗与其在不同计算机上的实现有关。不同版本的 Python 有可能使用不同方法实现同一种数据类型。

确定一个 Python 程序内存使用的典型方法是先统计程序使用的对象的数量,然后根据对象的类型乘以各对象占用的字节数。使用函数 sys.getsizeof(x)可以返回一个内置数据类型 x 在系统中所占用的字节数。

【例 11.2】 Python 语言中对象占用内存大小示例。

```
>>> import sys
>>> sys.getsizeof(100)                    #整数对象占用内存大小.输出:28
>>> sys.getsizeof("1.23")                 #字符串对象占用内存大小.输出:53
>>> sys.getsizeof(True)                   #布尔逻辑型对象占用内存大小.输出:28
```

11.2　查找算法

11.2.1　顺序查找法

查找算法是在程序设计中最常用到的算法。假定要从 n 个元素中查找 x 的值是否存在,最原始的方法是从头到尾逐个查找,这种查找方法称为顺序查找法。

顺序查找法有 3 种情形可能发生:在最好的情况下,第一项就是要找的数据对象,只有一次比较;在最差的情况下,需要 n 次比较,其全部比较完之后查不到数据;在平均情况下,比较次数为 n/2 次。即算法的时间复杂度为 O(n)。

【例 11.3】 在列表中顺序查找特定数值 x(sequentialSearch.py)。

```
def sequentialSearch(alist, item):        #顺序查找法
    pos = 0                               #初始查找位置
```

```
    found = False                                        #未找到数据对象
    while pos < len(alist) and not found:                #列表未结束并且还未找到则一直循环
        if alist[pos] == item:                           #找到匹配对象,返回 True
            found = True
        else:                                            #否则查找位置 +1
            pos = pos+1
    return found
def main():
    testlist = [1, 3, 33, 8, 37, 29, 32, 15, 5]          #测试数据列表
    print(sequentialSearch(testlist, 3))                 #查找数据 3
    print(sequentialSearch(testlist, 13))                #查找数据 13
if __name__ == '__main__': main()
```

程序运行结果如下。

```
True
False
```

【例 11.4】 在列表中顺序查找最大值和最小值(MaxMin.py)。

```
def max1(alist):                                         #查找最大值
    pos = 0                                              #初始查找位置
    iMax = alist[0]                                      #假设第一个值最大
    while pos < len(alist):                              #在列表中循环
        if alist[pos] > iMax:                            #如果列表当前值大于最大值 iMax
            iMax = alist[pos]                            #则当前值为最大值 iMax
        pos = pos+1                                      #查找位置 +1
    return iMax                                          #返回最大值
def min1(alist):                                         #查找最小值
    iMin = alist[0]                                      #假设第一个值最小
    for item in alist:                                   #对于列表中的每个数值
        if item < iMin:                                  #如果列表当前值小于最小值 iMin
            iMin = item                                  #则当前值为最小值 iMin
    return iMin                                          #返回最小值
def main():
    testlist = [1, 3, 33, 8, 37, 29, 32, 15, 5]          #测试数据列表
    print("最大值 =",max1(testlist))                      #查找并打印列表中的最大值
    print("最小值 =",min1(testlist))                      #查找并打印列表中的最小值
if __name__ == '__main__': main()
```

程序运行结果如下。

```
最大值 = 37
最小值 = 1
```

11.2.2 二分查找法

二分查找法又称折半查找法,用于预排序列表的查找问题。

如果要在排序列表 alist 中查找元素 t,首先将列表 alist 中间位置的项与查找关键字 t 比较,如果两者相等,则查找成功;否则利用中间项将列表分成前、后两个子表,如果中间位置项目大于 t,则进一步查找前一子表,否则进一步查找后一子表。重复以上过程,直到找到满足条件的记录,即查找成功;或者直到子表不存在为止,即查找不成功。

对于包含 N 个元素的表,其时间复杂度为 $O(\log_2 N)$。

【例 11.5】　二分查找法的递归实现(binarySearch.py)。

```
def _binarySearch(key, a, lo, hi):
    if hi <= lo: return -1                       #查找失败,返回-1
    mid = (lo + hi) // 2                         #计算中间位置
    if a[mid] > key:                             #中间位置项目大于查找关键字
        return _binarySearch(key, a, lo, mid)    #递归查找前一子表
    elif a[mid] < key:                           #中间位置项目小于查找关键字
        return _binarySearch(key, a, mid+1, hi)  #递归查找后一子表
    else:                                        #中间位置项目等于查找关键字
        return mid                               #查找成功,返回下标位置
def binarySearch(key, a):                        #二分查找
    return _binarySearch(key, a, 0, len(a))      #递归二分查找法
def main():
    a = [1,13,26,33,45,55,68,72,83,99]
    print("关键字位于列表索引",binarySearch(33, a)) #二分查找关键字 33
    print("关键字位于列表索引",binarySearch(58, a)) #二分查找关键字 58
if __name__ == '__main__': main()
```

程序运行结果如下。

```
关键字位于列表索引 3
关键字位于列表索引 -1
```

【例 11.6】　二分查找法的非递归实现(binarySearchNoRecursion.py)。

```
def binarySearch(key, a):                        #二分查找法的非递归实现
    low = 0                                      #左边界
    high = len(a) - 1                            #右边界
    while low <= high:                           #左边界小于等于右边界,则循环
        mid = (low + high) // 2                  #计算中间位置
        if a[mid] < key:                         #中间位置项目小于查找关键字
            low = mid + 1                        #调整左边界(在后一子表查找)
        elif a[mid] > key:                       #中间位置项目大于查找关键字
            high = mid - 1                       #调整右边界(在前一子表查找)
        else:                                    #中间位置项目等于查找关键字
            return mid                           #查找成功,返回下标位置
        return -1                                #查找不成功(不存在关键字),返回-1
def main():
    a = [1,13,26,33,45,55,68,72,83,99]
    print("关键字位于列表索引",binarySearch(33, a)) #二分查找关键字 33
    print("关键字位于列表索引",binarySearch(58, a)) #二分查找关键字 58
if __name__ == '__main__': main()
```

11.2.3　Python 语言提供的查找算法

Python 语言提供了下列查找算法。

(1) 运算符 in:"x in alist"测试值 x 是否在列表 alist 中存在。

(2) 内置函数 max()、min():查找列表的最大值和最小值。

【例 11.7】　Python 语言提供的查找算法示例。

```
>>> 3 in [1, 3, 33, 8, 37, 29, 32, 15, 5]        #输出:True
>>> 13 in [1, 3, 33, 8, 37, 29, 32, 15, 5]       #输出:False
>>> max([1, 3, 33, 8, 37, 29, 32, 15, 5])        #输出:37
>>> min([1, 3, 33, 8, 37, 29, 32, 15, 5])        #输出:1
```

11.3 排序算法

11.3.1 冒泡排序法

冒泡排序法是最简单的排序算法。对于包含 N 个元素的列表 A,按递增顺序排序的冒泡法的算法如下。

(1) 第 1 轮比较:从第一个元素开始,对列表中的 N 个元素进行两两大小比较,如果不满足升序关系,则交换。即 A[0]与 A[1]比较,若 A[0]>A[1],则 A[0]与 A[1]交换;然后 A[1]与 A[2]比较,若 A[1]>A[2],则 A[1]与 A[2]交换;直到最后 A[N-2]与 A[N-1]比较,若 A[N-2]>A[N-1],则 A[N-2]与 A[N-1]交换。第 1 轮比较完成后,列表元素中最大的数"沉"到列表最后,而较小的数如同气泡一样上浮一个位置,因此称为"冒泡法"排序。

(2) 第 2 轮比较:从第一个元素开始,对列表中的前 N-1 个元素(第 N 个元素,即 A[N-1]已经最大,无须参加排序)继续两两大小比较,如果不满足升序关系,则交换。第 2 轮比较完成后,列表元素中次大的数"沉"到最后,即 A[N-2]为列表元素中次大的数。

(3) 依此类推,进行第 N-1 轮比较后,列表中的所有元素均按递增顺序排好。

若要按递减顺序对列表排序,则每次进行两两大小比较时,如果不满足降序关系,则交换即可。

冒泡排序法的过程如表 11-3 所示。

表 11-3 冒泡排序法示例

原始列表	2	97	86	64	50	80	3	71	8	76
第 1 轮比较	2	86	64	50	80	3	71	8	76	**97**
第 2 轮比较	2	64	50	80	3	71	8	76	**86**	**97**
第 3 轮比较	2	50	64	3	71	8	76	**80**	**86**	**97**
第 4 轮比较	2	50	3	64	8	71	**76**	**80**	**86**	**97**
第 5 轮比较	2	3	50	8	64	**71**	**76**	**80**	**86**	**97**
第 6 轮比较	2	3	8	50	**64**	**71**	**76**	**80**	**86**	**97**
第 7 轮比较	2	3	8	**50**	**64**	**71**	**76**	**80**	**86**	**97**
第 8 轮比较	2	3	**8**	**50**	**64**	**71**	**76**	**80**	**86**	**97**
第 9 轮比较	2	**3**	**8**	**50**	**64**	**71**	**76**	**80**	**86**	**97**

冒泡排序法的主要时间消耗是比较次数。当 i=1 时,比较次数为 N-1;当 i=2 时,比较次数为 N-2;依此类推,总比较次数为(N-1)+(N-2)+…+2+1=N(N-1)/2,故冒泡排序法的时间复杂度为 $O(N^2)$。

【例 11.8】 冒泡排序法的实现(bubbleSort.py)。

```
def bubbleSort(a):
    for i in range(len(a) - 1, - 1, - 1):          # 外循环
        for j in range(i):                          # 内循环
            if a[j] > a[j + 1]:                     # 大数往下沉
                a[j], a[j + 1] = a[j + 1], a[j]
        # print(a)                                  # 跟踪调试
def main():
    a = [2,97,86,64,50,80,3,71,8,76]
```

```
    bubbleSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[2, 3, 8, 50, 64, 71, 76, 80, 86, 97]
```

11.3.2 选择排序法

对于包含 N 个元素的列表 A,按递增顺序排序的选择法的基本思想是每次在若干无序数据中查找最小数,并放在无序数据中的首位。其算法如下:

(1) 从 N 个元素的列表中找最小值及其下标,最小值与列表的第 1 个元素交换。

(2) 从列表的第 2 个元素开始的 N−1 个元素中再找最小值及其下标,该最小值(即整个列表元素的次小值)与列表的第 2 个元素交换。

(3) 依此类推,进行第 N−1 轮选择和交换后,列表中的所有元素均按递增顺序排好。

若要按递减顺序对列表排序,只要每次查找并交换最大值即可。

选择排序法的过程如表 11-4 所示。

表 11-4 选择排序法示例

原始数组	59	12	77	64	72	69	46	89	31	9
第 1 轮比较	**9**	12	77	64	72	69	46	89	31	59
第 2 轮比较	**9**	**12**	77	64	72	69	46	89	31	59
第 3 轮比较	**9**	**12**	**31**	64	72	69	46	89	77	59
第 4 轮比较	**9**	**12**	**31**	**46**	72	69	64	89	77	59
第 5 轮比较	**9**	**12**	**31**	**46**	**59**	69	64	89	77	72
第 6 轮比较	**9**	**12**	**31**	**46**	**59**	**64**	69	89	77	72
第 7 轮比较	**9**	**12**	**31**	**46**	**59**	**64**	**69**	89	77	72
第 8 轮比较	**9**	**12**	**31**	**46**	**59**	**64**	**69**	**72**	77	89
第 9 轮比较	**9**	**12**	**31**	**46**	**59**	**64**	**69**	**72**	**77**	89

选择排序法的主要时间消耗是比较次数。当 $i=1$ 时,比较次数为 $N-1$;当 $i=2$ 时,比较次数为 $N-2$;依此类推,总比较次数为 $(N-1)+(N-2)+\cdots+2+1=N(N-1)/2$,故选择排序法的时间复杂度为 $O(N^2)$。

【例 11.9】 选择排序法的实现(selectionSort.py)。

```
def selectionSort(a):
    for i in range(0, len(a)):                  #外循环(0～N-1)
        m = i                                   #当前位置下标
        for j in range(i + 1, len(a)):          #内循环
            if a[j] < a[m]:                     #查找最小值的位置
                m = j
        a[i], a[m] = a[m], a[i]                 #元素交换
        #print(a)                               #跟踪调试
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    selectionSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

11.3.3 插入排序法

对于包含 N 个元素的列表 A，按递增顺序排序的插入排序法的基本思想是依次检查列表中的每个元素，将其插入到其左侧已经排好序的列表中的适当位置。其算法如下：

（1）第 2 个元素与列表中其左侧的第 1 个元素比较，如果 A[0]＞A[1]，则交换位置，结果左侧的两个元素排序完毕。

（2）第 3 个元素依次与其左侧的列表的元素比较，直到插入对应的排序位置，结果左侧的 3 个元素排序完毕。

（3）以此类推，进行第 N－1 轮比较和交换后，列表中的所有元素均按递增顺序排好。

若要按递减顺序对列表排序，只要每次查找并交换最大值即可。

插入排序法的过程如表 11-5 所示。

表 11-5 插入排序法示例

原始数组	59	12	77	64	72	69	46	89	31	9
第 1 轮比较	**12**	59	77	64	72	69	46	89	31	9
第 2 轮比较	12	59	77	64	72	69	46	89	31	9
第 3 轮比较	12	59	**64**	77	72	69	46	89	31	9
第 4 轮比较	12	59	64	**72**	77	69	46	89	31	9
第 5 轮比较	12	59	64	**69**	72	77	46	89	31	9
第 6 轮比较	12	**46**	59	64	69	72	77	89	31	9
第 7 轮比较	12	46	59	64	69	72	77	89	31	9
第 8 轮比较	12	**31**	46	59	64	69	72	77	89	9
第 9 轮比较	**9**	12	31	46	59	64	69	72	77	89

在最理想的情况下（列表处于排序状态），while 循环仅需要比较一次，故总的运行时间为线性；在最差的情况下（列表为逆序状态），内循环指令的执行次数为 $1+2+\cdots+N-1=N(N-1)/2$，故插入排序法的时间复杂度为 $O(N^2)$。

【例 11.10】 插入排序法的实现（insertSort.py）。

```
def insertSort(a):
    for i in range(1, len(a)):                    #外循环(1～N-1)
        j = i
        while (j > 0) and (a[j] < a[j-1]):        #内循环
            a[j], a[j-1] = a[j-1], a[j]           #元素交换
            j -= 1                                #继续循环
        #print(a)                                 #跟踪调试
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    insertSort(a)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

11.3.4　归并排序法

归并排序法基于分而治之(Divide and Conquer)的思想。算法的操作步骤如下。

(1) 将包含 N 个元素的列表分成两个含 N/2 元素的子列表。

(2) 对两个子列表递归调用归并排序(最后可以将整个列表分解成 N 个子列表)。

(3) 合并两个已排好序的子列表。

假设列表 a = [59,12,77,64,72,69,46,89,31,9],归并排序法的示意图如图 11-2 所示。

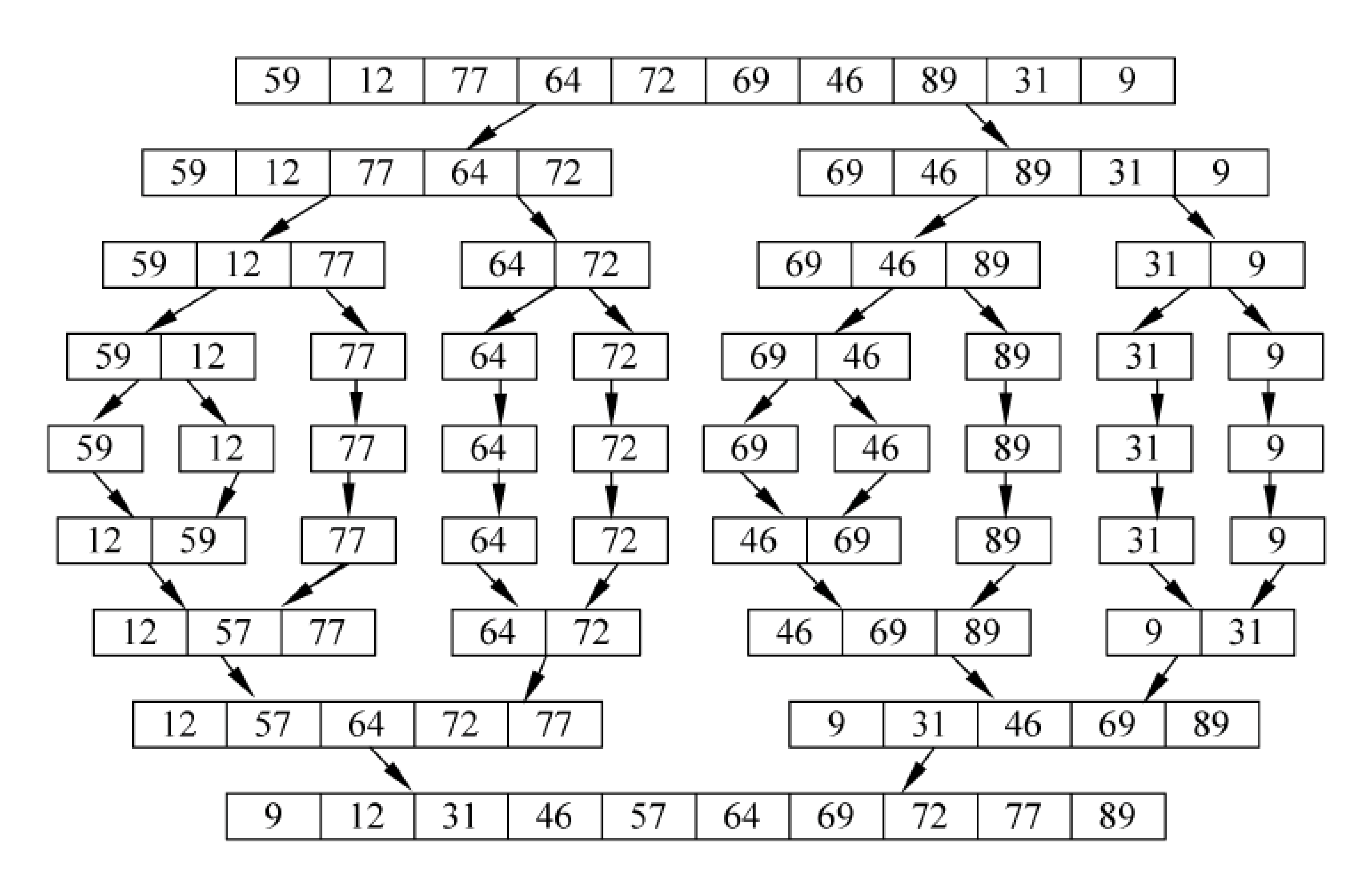


图 11-2　归并排序法示意图

对于长度为 N 的列表,归并排序法将列表分成子列表一共要 $\log_2 N$ 步,每步都是一个合并有序列表的过程,时间复杂度可以记为 O(N),故归并排序法的时间复杂度为 $O(N\log_2 N)$。其效率是比较高的。

【例 11.11】 归并排序法的实现(mergeSort.py)。

```
def merge(left, right):                                  #合并两个列表
    merged = []
    i, j = 0, 0                                          #i 和 j 分别作为 left 和 right 的下标
    left_len, right_len = len(left), len(right)          #分别获取左、右子列表的长度
    while i < left_len and j < right_len:                #循环归并左、右子列表的元素
        if left[i] <= right[j]:
            merged.append(left[i])                       #归并左子列表的元素
            i += 1
        else:
            merged.append(right[j])                      #归并右子列表的元素
            j += 1
    merged.extend(left[i:])                              #归并左子列表的剩余元素
    merged.extend(right[j:])                             #归并右子列表的剩余元素
    #print(left,right,merged)                            #跟踪调试
    return merged                                        #返回归并好的列表
def mergeSort(a):                                        #归并排序
    if len(a) <= 1:                                      #空或者只有 1 个元素,直接返回列表
        return a
    mid = len(a) // 2                                    #列表中间位置
    left = mergeSort(a[:mid])                            #归并排序左子列表
    right = mergeSort(a[mid:])                           #归并排序右子列表
```

```
    return merge(left, right)                    #合并排好序的左、右两个子列表
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    a1 = mergeSort(a)
    print(a1)
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

11.3.5 快速排序法

快速排序是对冒泡排序的一种改进,由 C. A. R. Hoare 在 1962 年提出。其基本思想是通过一趟排序将要排序的数据分割成独立的两部分,其中一部分的所有数据比另外一部分的所有数据都要小,然后对这两部分数据分别进行快速排序。

快速排序法的一趟排序的操作步骤如下。

(1) 设置两个变量 i 和 j,分别为列表首、末元素的下标,即 i=0,j=N-1。

(2) 设置列表的第一个元素为关键数据,即 key=A[0]。

(3) 从 j 开始向前搜索,找到第一个小于 key 的值 A[j],将 A[j]和 A[i]互换。

(4) 从 i 开始向后搜索,找到第一个大于 key 的 A[i],将 A[i]和 A[j]互换。

(5) 重复第(3)和(4)步,直到 i=j。

假设列表 a = [59,12,77,64,72,69,46,89,31,9],快速排序法的一趟排序示例如表 11-6 所示。

表 11-6 快速排序法的一趟排序示例

下 标	0	1	2	3	4	5	6	7	8	9
原始数组 i=0, j=9, key=59	**59**	12	77	64	72	69	46	89	31	**9**
第 1 轮比较交换 i=0, j=9	**9**	12	77	64	72	69	46	89	31	**59**
第 2 轮比较交换 i=2, j=9	9	12	**59**	64	72	69	46	89	31	**77**
第 3 轮比较交换 i=2, j=8	9	12	**31**	64	72	69	46	89	**59**	77
第 4 轮比较交换 i=3, j=8	9	12	31	**59**	72	69	46	89	**64**	77
第 5 轮比较交换 i=3, j=6	9	12	31	**46**	72	69	**59**	89	64	77
第 6 轮比较交换 i=4, j=6	9	12	31	46	**59**	69	**72**	89	64	77
第 7 轮比较交换 i=4, j=4	9	12	31	46	**59**	69	72	89	64	77

快速排序在最坏情况下,每次划分选取的基准都是当前无序列表中关键字最小(或最大)的记录,时间复杂度为 $O(N^2)$;在平均情况下,其时间复杂度为 $O(N\log_2 N)$。

【例 11.12】 快速排序法的实现(quickSort. py)。

```
def quickSort(a, low, high):               #对列表 a 快速排序,列表下界为 low、上界为 high
    i = low                                #i 等于列表下界
    j = high                               #j 等于列表上界
    if i >= j:                             #如果下界大于等于上界,返回结果列表 a
        return a
    key = a[i]                             #设置列表的第一个元素为关键数据
    #print(key)                            #跟踪调试
    while i < j:                           #循环直到 i = j
        while i < j and a[j] >= key:       #j 开始向前搜索,找到第一个小于 key 的值 a[j]
            j = j - 1
```

```
            a[i] = a[j]
            while i < j and a[i] <= key:      #i开始向后搜索,找到第一个大于key的值a[i]
            i = i+1
            a[j] = a[i]
        a[i] = key                            #a[i]等于关键数据
        #print(a)                             #跟踪调试
        quickSort(a, low, i-1)                #递归调用快速排序法(列表下界为low、上界为i-1)
        quickSort(a, j+1, high)               #递归调用快速排序法(列表下界为j+1、上界为high)
def main():
    a = [59,12,77,64,72,69,46,89,31,9]
    quickSort(a, 0, len(a)-1)
    print(a)
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

11.3.6 Python 语言提供的排序算法

Python 语言提供了下列排序算法。

(1) 内置数据类型 list 中的方法 sort():把列表的项按升序重新排列。

(2) 内置函数 sorted():保持原列表不变,函数返回一个新的包含按升序排列的项的列表。

Python 系统排序方法使用了一种归并排序算法的版本,但使用了 Python 无法编写的底层实现,从而避免了 Python 本身附加的大量开销,故其速度比 mergeSort.py 快很多(10～20倍)。系统排序方法同样能够用于任何可比较的数据类型,例如 Python 内置的 str、int 和 float 数据类型。

【例 11.13】 Python 语言提供的查找算法示例。

```
>>> a = [59,12,77,64,72,69,46,89,31,9]
>>> sorted(a)                  #输出:[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
>>> a                          #输出:[59, 12, 77, 64, 72, 69, 46, 89, 31, 9]
>>> a.sort()
>>> a                          #输出:[9, 12, 31, 46, 59, 64, 69, 72, 77, 89]
```

11.4 常用数据结构

11.4.1 数据结构概述

数据结构是计算机存储、组织数据的方式,通常由数据元素的集合和集合中数据元素之间的关系组成。算法的实现基于数据结构,选择恰当的数据结构可以带来更高的运行或者存储效率。

数据结构通常由 3 个部分组成,即数据的逻辑结构、数据的物理结构和数据的运算结构。

(1) 数据的逻辑结构:数据的逻辑结构反映数据元素之间的逻辑关系。数据的逻辑结构主要包括线性结构(一对一的关系)、树形结构(一对多的关系)、图形结构(多对多的关系)、集合。

(2) 数据的物理结构:数据的物理结构反映数据的逻辑结构在计算机存储空间的存放形式,即数据结构在计算机中的表示。其具体实现的方法包括顺序、链接、索引、散列等多种形式。一种数据结构可以由一种或多种物理存储结构实现。

(3) 数据的运算结构：数据的运算结构反映在数据的逻辑结构上定义的操作算法，例如检索、插入、删除、更新和排序等。

11.4.2 常用数据结构概述

计算机中包括以下几种常用的数据结构。

(1) 数组：按序排列的同类数据元素的集合。

(2) 线性表：排列在一条线上或一个环上的数据元素(线性关系)。

(3) 栈：遵循先进后出(FILO)原则，只允许在某一端插入和删除的线性表。

(4) 队列：遵循先进先出(FIFO)原则，只允许在表的前端进行删除操作、在表的后端进行插入的线性表。

(5) 链表：链表由一系列结点(元素)组成，每个结点包括两个部分，即数据域和指针域。

(6) 树：由 n(n≥1)个有限结点组成的一个具有层次关系的集合，其形状像一个倒挂的树。

(7) 图：图是由顶点集合 V(Vertex)和边集合 E(Edge)组成的，定义为 G=(V,E)。

(8) 堆：堆(heap)是一个树形数据结构，其中子结点与父结点是一种有序关系。

(9) 散列表：散列表(Hash table，也叫哈希表)是把键值映射(映射函数)到表(散列表)的数据结构，通过键值可以实现快速查找。

11.4.3 Python 中的 collections 模块

Python 中的 collections 模块是 Python 标准模块，实现了许多常用的数据结构。

collections.abc 模块包含若干 ABCs(Abstract Base Classes，抽象基类)。抽象基类用于定义接口，继承抽象基类的派生类实现这些接口功能。抽象类的派生类需要实现对应的抽象方法；抽象类的派生类自动拥有抽象类包含的继承方法(mixins)，不需要重新实现，方便派生类的开发。

collections 模块和容器类型包含若干实用的容器类型对象，用于实现常用的数据结构。例如 collections.deque 实现了线程安全的双端队列，等等。

11.5 数　组

Python 语言没有直接提供数组数据类型，通常使用列表作为数组，或者使用标准库 array 模块中的 array 对象作为数组。如果要实现高效的数值计算，通常使用第三方科学计算模块 NumPy 来实现数组(参见 14.5 节)。

11.5.1 列表和数组

列表支持数组要求的 4 种核心操作，即创建数组、索引访问、索引赋值和迭代遍历。

【例 11.14】 Python 数组示例(deck.py)：生成一副扑克牌(每副扑克牌包含 52 张牌，大、小王除外)，并且随机洗牌后输出一副扑克牌。

```
import random
SUITS = ['Club', 'Diamond', 'Heart', 'Spade']      #梅花、方块、红桃、黑桃
RANKS = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
#生成一副扑克牌，每副扑克牌包含 52 张牌(大、小王除外)
deck = []                                          #一副扑克牌
```

```
for rank in RANKS:
    for suit in SUITS:
        card = rank + ' of ' + suit
        deck += [card]
#洗牌
n = len(deck)
for i in range(n):
    r = random.randrange(i, n)
    temp = deck[r]
    deck[r] = deck[i]
    deck[i] = temp
#输出一副扑克牌
for s in deck: print(s)
```

程序运行结果(部分,并且每次随机)如下:

```
4 of Club
3 of Diamond
5 of Diamond
A of Spade
9 of Club
4 of Diamond
7 of Diamond
4 of Heart
K of Club
7 of Heart
…
```

11.5.2 array.array 对象和数组

array 模块包含一个 array 对象,用于实现其他编程语言中的数组数据结构。array 对象是包含相同基本数据类型的列表,其操作与 list 对象基本一致,区别是创建 array 对象时必须指定其元素类型 typecode,且其元素只能为该类型,否则将导致 TypeError。

array 对象的构造函数如下。

array(typecode[, initializer])

其中,typecode 为 array 对象中数据元素的类型;initializer 为初始化数据序列或可迭代对象,其元素类型必须与 typecode 一致。

array 对象支持包括索引访问、切片操作、连接操作、重复操作、成员关系操作、比较运算操作,以及求长度、最大值、最小值等的基本操作。

和 list 对象类似,array 是可变对象,故可以改变其元素的值,也可以通过 del 删除某元素;可以改变其切片的值,也可以通过 del 删除切片。

【例 11.15】 array.array 对象和数组示例。

```
>>> import array
>>> a = array.array('b', (1, 2, 3, 4, 5))
>>> a[1]                          #输出:2
>>> a[1] = 22
>>> a[1:]                         #输出:array('b', [22, 3, 4, 5])
>>> a[0] = "abc"                  #报错.TypeError: an integer is required (got type str)
```

11.6 栈和队列

队列(Queue)是先进先出的序列(First In First Out,FIFO),即最先添加的元素是最先弹出的元素;栈(Stack)是后进先出的队列(Last In First Out,LIFO),即最后添加(push)的元素是最先弹出(pop)的元素。

11.6.1 栈的实现:使用列表

向列表最后位置添加元素和从最后位置移除元素非常方便和高效,故使用 list 可以快捷、高效地实现栈。list.append()方法对应于入栈操作(push);list.pop()对应于出栈操作(pop)。

列表可以实现队列(Queue),但并不适合。因为从列表的头部移除一个元素,列表中的所有元素都需要移动位置,所以效率不高,此时可以使用 collections 模块中的 deque 对象。

【例 11.16】 栈的实现示例(stack.py):创建一个包含整数 1 和 2 的栈,展示入栈和出栈操作,以及打印栈的内容。

```
class Stack:
    def __init__(self,size = 16):          #初始化栈
        self.stack = []
    def push(self,obj):                    #入栈操作(push)
        self.stack.append(obj)
    def pop(self):                         #出栈操作(pop)
        try:
            return self.stack.pop()
        except IndexError as e:
            print("stack is empty")
    def __str__(self):
        return str(self.stack)
def main():
    stack = Stack()                        #创建并初始化栈
    stack.push(1)                          #整数 1 入栈
    stack.push(2)                          #整数 2 入栈
    print(stack)                           #打印栈的内容
    stack.pop()                            #整数 2 出栈
    stack.pop()                            #整数 1 出栈
    stack.pop()                            #出栈操作,但因为是空栈,提示"stack is empty"
if __name__ == '__main__': main()
```

程序运行结果如下。

```
[1, 2]
stack is empty
```

11.6.2 deque 对象

collections.deque(双端队列)支持从任意一端增加和删除元素。deque 是线程安全的、内存高效的队列,它被设计为从两端追加和弹出都非常快。

```
deque([iterable[, maxlen]])              #构造函数
```

其中,可选的 iterable 为初始元素;maxlen 用于指定队列长度(默认无限制)。

deque 对象 dq 支持下列方法。

- dq.append(x)：在右端添加元素 x。
- dq.appendleft(x)：在左端添加元素 x。
- dq.pop()：从右端弹出元素。若队列中无元素，则导致 IndexError。
- dq.popleft()：从左端弹出元素。若队列中无元素，则导致 IndexError。
- dq.extend(iterable)：在右端添加序列 iterable 中的元素。
- dq.extendleft(iterable)：在左端添加序列 iterable 中的元素。
- dq.remove(value)：移除第一个找到的 x。若未找到，则导致 IndexError。
- dq.count(x)：返回元素 x 在队列中出现的个数。
- dq.clear()：删除所有元素，即清空队列。
- dq.reverse()：反转队列中的所有元素。
- dq.rotate(n)：如果 n>0，所有元素向右移动 n 个位置(循环)，否则向左。

【例 11.17】 deque 对象示例(deque_tail.py)：读取文件，返回文件最后的 n 行内容。相当于执行 Unix 中的 tail 命令。

```
import collections
def tail(filename, n = 10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return collections.deque(f, n)
if __name__ == '__main__':
    path = r'deque_tail.py'
    dq = tail(path, n = 2)                    #最后两行
    print(dq.popleft())
    print(dq.popleft())
```

程序运行结果如下。

```
    print(dq.popleft())
    print(dq.popleft())
```

11.6.3　deque 作为栈

deque 对象方法 append()用于入栈操作；pop()用于出栈操作。

【例 11.18】 deque 作为栈示例。

```
>>> from collections import *
>>> dq = deque()
>>> dq.append(1); dq.append(2); dq.append(3)          #整数 1、2、3 入栈
>>> dq.pop(); dq.pop(); dq.pop()                      #整数 3、2、1 出栈
```

11.6.4　deque 作为队列

deque 对象方法 append()用于进队操作；popleft()用于出队操作。

【例 11.19】 deque 作为队列示例。

```
>>> from collections import *
>>> dq = deque()
>>> dq.append(1); dq.append(2); dq.append(3)          #整数 1、2、3 入队
>>> dq.popleft(); dq.popleft(); dq.popleft()          #整数 1、2、3 出队
```

11.7 集　　合

集合数据类型是没有顺序的简单对象的聚集,且集合中的元素不重复。Python 集合数据类型包括可变集合对象(set)和不可变集合对象(frozenset)。

11.7.1 集合的定义

可变集合(set)通过花括号中用逗号分隔的项目定义。其基本形式如下:

```
{x1[, x2, …, xn]}
```

其中,x_1,x_2,…,x_n 为任意可 hash 对象。集合中的元素不可重复,且无序,其存储依据对象的 hash 码。hash 码是根据对象的值计算出来的一个唯一值。一个对象如果定义了特殊方法__hash__(),则该对象为可 hash 对象。所有内置不可变对象(bool、int、float、complex、str、tuple、frozenset 等)都是可 hash 对象;所有内置可变对象(list、dict、set)都是非 hash 对象(因为可变对象的值可以变化,故无法计算一个唯一的 hash 值)。在集合中可以包含内置不可变对象,不能包含内置可变对象。

注意:{}表示空的 dict,因为 dict 也使用花括号定义。空集为 set()。

可变集合也可以通过创建 set 对象来创建,不可变集合通过创建 frozenset 对象来创建。其基本形式如下:

- **set()**　　　　　　　#创建一个空的可变集合
- **set(iterable)**　　　#创建一个可变集合,包含的项目为可枚举对象 iterable 中的元素
- **frozenset()**　　　　#创建一个空的不可变集合
- **frozenset(iterable)**　#创建一个不可变集合,包含的项目为可枚举对象 iterable 中的元素

【例 11.20】 创建集合对象示例。

>>> {1,2,1}
{1, 2}
>>> {1,'a',True}
{1, 'a'}
>>> {1.2, True}
{True, 1.2}
	>>> set()
	set()
	>>> frozenset()
	frozenset()
	>>> set('Hello')
	{'H', 'e', 'o', 'l'}
		>>> {'a',[1,2]}
		Traceback(most recent call last):
		  File "<pyshell #13>", line 1, in <module>
		{'a',[1,2]}
		TypeError: unhashable type: 'list'

11.7.2 集合解析表达式

使用集合解析表达式可以简单、高效地处理一个可迭代对象,并生成结果集合。集合解析表达式的形式如下:

- **{expr for i_1 in 序列 1… for i_N in 序列 N}**　　#迭代序列中的所有内容,并计算生成字典
- **{expr for i_1 in 序列 1… for i_N in 序列 N if cond_expr}**　#按条件迭代,并计算生成字典

表达式 expr 使用每次迭代内容 i_1~i_N 计算生成一个集合。如果指定了条件表达式 cond_expr,则只有满足条件的元素参与迭代。

【例 11.21】 集合解析表达式示例。

```
>>> {i for i in range(5)}              #输出:{0, 1, 2, 3, 4}
>>> {2 * * i for i in range(5)}        #输出:{1, 2, 4, 8, 16}
>>> {x * * 2 for x in [1, 1, 2]}       #输出:{1, 4}
```

11.7.3 判断集合元素是否存在

用户可以通过下列方式之一判断元素 x 是否在集合 s 中存在：

```
x in s                                        #如果为 True,则表示存在
x not in s                                    #如果为 True,则表示不存在
```

【例 11.22】 集合中元素的判断示例。

```
>>> s = set('Hello')
>>> s
{'H', 'e', 'o', 'l'}
```

```
>>> 'h' in s
False
>>> 'o' not in s
False
```

11.7.4 集合的运算：并集、交集、差集和对称差集

集合支持表 11-7 所示的集合运算。

表 11-7 集合运算

运 算 符	说 明
s1 | s2 | …	返回 s1、s2……的并集：s1∪s2∪…
s1 & s2 & …	返回 s1、s2……的交集：s1∩s2∩…
s1 - s2 - …	返回 s1、s2……的差集，也记作 s1\s2\…
s1 ^ s2	返回 s1、s2 的对称差集：s1△s2

集合的对象方法如表 11-8 所示。

表 11-8 集合的对象方法

方 法	说 明
s1.isdisjoint(s2)	如果集合 s1 和 s2 没有共同元素，返回 True；否则返回 False
s1.issubset(s2)	如果集合 s1 是 s2 的子集，返回 True；否则返回 False
s1.issuperset(s2)	如果集合 s1 是 s2 的超集，返回 True；否则返回 False
s1.union(s2，…)	返回 s1、s2……的并集：s1∪s2∪…
s1.intersection(s2，…)	返回 s1、s2……的交集：s1∩s2∩…
s1.difference(s2，…)	返回 s1、s2……的差集：s1－s2－…
s1.symmetric_difference(s2)	返回 s1 和 s2 的对称差集：s1△s2

【例 11.23】 集合的运算示例。

```
>>> s1 = {1,2,3}
>>> s2 = {2,3,4}
>>> s1 | s2
{1, 2, 3, 4}
>>> s1 & s2
{2, 3}
```

```
>>> s1 - s2
{1}
>>> s1 ^ s2
{1, 4}
>>> s1.union(s2)
{1, 2, 3, 4}
```

```
>>> s1.intersection(s2)
{2, 3}
>>> s1.difference(s2)
{1}
>>> s1.symmetric_difference(s2)
{1, 4}
```

11.7.5 集合的比较运算：相等、子集和超集

集合支持表 11-9 所示的比较运算。

表 11-9 集合的比较运算

运算符	说明	运算符	说明
s1 == s2	s1 和 s2 的元素相同	s1 <= s2	s1 是 s2 的子集
s1 != s2	s1 和 s2 的元素不完全相同	s1 >= s2	s1 是 s2 的超集
s1 < s2	s1 是 s2 的纯子集	s1 > s2	s1 是 s2 的纯超集

集合对象的比较方法包括 s1.isdisjoint(s2)、s1.issubset(s2)和 s1.issuperset(s2)，参见表 11-8。

【例 11.24】 集合比较运算示例。

```
>>> s1 = {1,2,3}
>>> s2 = {3,2,1}
>>> s3 = {1,2}
>>> s4 = {7,9}
>>> s1 == s2
True
>>> s1!= s4
True
>>> s3 <= s1
True
>>> s2 > s3
True
>>> s3 < s4
False
>>> s3 > s4
False
>>> s1 >= s2
True
>>> s1.isdisjoint(s2)
False
>>> s3.issubset(s1)
True
>>> s2.issuperset(s3)
True
```

11.7.6 集合的长度、最大值、最小值、元素和

通过内置函数 len()、max()、min()、sum()可以获取集合的长度、元素最大值、元素最小值、元素和。如果元素有非整数，则求和将导致 TypeError。

【例 11.25】 集合的长度、最大值、最小值、元素和示例。

```
>>> s1 = {1,3,5,7,9}
>>> s2 = {'1','2','3'}
>>> len(s1)
5
>>> max(s1)
9
>>> min(s2)
'1'
>>> sum(s2)
Traceback(most recent call last):
  File "<pyshell #16>", line 1, in <module>
    sum(s2)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

11.7.7 可变集合的方法

set 集合是可变对象，包含的主要方法如表 11-10 所示。假设该表中的示例基于“s1={1,2,3}；s2={2,3,4}”。

表 11-10 可变集合对象的主要方法

方法	说明	示例
s1.update(s2,…) s1 |=s2 | …	并集 s1= s1∪s2∪…	>>> s1.update(s2) #s1={1, 2, 3, 4}
s1.intersection_update(s2,…) s1 &= s2 &…	交集 s1= s1∩s2∩…	>>> s1.intersection_update(s2) # s1={2, 3}
s1.difference_update(s2,…) s1 -= s2 -…	差集 s1= s1-s2-…	>>> s1.difference_update(s2) # s1={1}

续表

方　法	说　明	示　例
s1. symmetric_difference_update(s2) s1 ^= s2	对称差集 s1= s1△s2	s1. symmetric_difference_update(s2) # s1={1, 4}
s. add(x)	把对象 x 添加到集合 s	>>> s1. add('a')　# s1={1, 2, 3, 'a'}
s. remove(x)	从集合 s 中移除对象 x。若不存在,则导致 KeyError	>>> s1. remove(1) # s1={2, 3}
s. discard(x)	从集合 s 中移除对象 x(如果存在)	>>> s1. discard(3) # s1={1, 2}
s. pop()	从集合 s 中随机弹出一个元素,如果 s 为空,则导致 KeyError	>>> s1. pop()　　# 输出:1。s1={2, 3}
s. clear()	清空集合 s	>>> s1. clear()　　# s1=set()

11.8　字　　典

字典(dict,或映射(map))是一组键/值对的数据结构。每个键对应于一个值。在字典中键不能重复,根据键可以查询到值。

11.8.1　对象的哈希值

字典是键和值的映射关系。字典的键必须是可 hash 的对象,即实现了__hash__()的对象。一个对象的 hash 值也可以使用内置函数 hash()获得。

【例 11.26】　对象的哈希(hash)值示例。

```
>>> hash(100)                  #结果:100
>>> hash(1.23)                 #结果:530343892119149569
>>> hash('abc')                #结果:901130859749610928
```

不可变对象 bool、int、float、complex、str、tuple、frozenset 等是可 hash 对象,可变对象通常是不可 hash 对象。不可变对象的内容可以改变,因此无法通过 hash()函数获取其 hash 值。

字典的键只能使用不可变对象,但字典的值可以使用不可变或可变对象。一般而言,应该使用简单的对象作为键。

11.8.2　字典的定义

字典通过花括号中用逗号分隔的项目(键/值。键/值对使用冒号分隔)定义。其基本形式如下:

```
{键 1:值 1[, 键 2:值 2, …, 键 n:值 n]}
```

键必须为可 hash 对象,因此不可变对象(bool、int、float、complex、str、tuple、frozenset 等)可以作为键;值可以为任意对象。字典中的键是唯一的,不能重复。

字典也可以通过创建 dict 对象来创建。其基本形式如下:

```
• dict()                       #创建一个空字典
• dict(**kwargs)               #使用关键字参数创建一个新的字典.此方法最紧凑
```

- **dict(mapping)**　　　　　　　　# 从一个字典对象创建一个新的字典
- **dict(iterable)**　　　　　　　　# 使用序列创建一个新的字典

【例 11.27】 创建字典对象示例。

>>> {}
{}
>>> {'a':'apple', 'b':'boy'}
{'a': 'apple', 'b': 'boy'}
>>> dict()
{}
	>>> dict({1:'food', 2:'drink'})
	{1: 'food', 2: 'drink'}
	>>> dict([('id','1001'),('name','Jenny')])
	{'id': '1001', 'name': 'Jenny'}
	>>> dict(baidu = 'baidu.com', google = 'google.com')
	{'baidu': 'baidu.com', 'google': 'google.com'}

11.8.3 字典的访问操作

字典 d 可以通过键 key 来访问，其基本形式如下：

- **d[key]**　　　　　　　　# 返回键为 key 的 value，如果 key 不存在，则导致 KeyError
- **d[key] = value**　　　　# 设置 d[key]的值为 value，如果 key 不存在，则添加键/值对
- **del d[key]**　　　　　　# 删除字典元素，如果 key 不存在，则导致 KeyError

【例 11.28】 字典的访问示例。

>>> d = {1:'food', 2:'drink'}
>>> d
{1: 'food', 2: 'drink'}
>>> d[1]
'food'
>>> d[3] = 'fruit'
>>> d
{1: 'food', 2: 'drink', 3: 'fruit'}
	>>> del d[2]
	>>> d
	{1: 'food', 3: 'fruit'}
	>>> d[2]
	Traceback(most recent call last):
	  File "<pyshell #100>", line 1, in <module>
	    d[2]
	KeyError: 2

11.8.4 字典的视图对象

字典 d 支持下列视图对象，通过它们可以动态访问字典的数据：

- **d.keys()**　　　　　　　# 返回字典 d 的键 key 的列表
- **d.values()**　　　　　　# 返回字典 d 的值 value 的列表
- **d.items()**　　　　　　　# 返回字典 d 的(key, value)对的列表

【例 11.29】 字典的视图对象示例。

>>> d = {1:'food', 2:'drink', 3:
'fruit'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> for k in d.keys():
    print(k, end = " ")
1 2 3
	>>> d.values()
	dict_values(['food', 'drink', '
	fruit'])
	>>> for v in d.values():
	    print(v, end = " ")
	food drink fruit
		>>> d.items()
		dict_items([(1, 'food'), (2, 'drink'
		), (3, 'fruit')])
		>>> for item in d.items():
		    print(item, end = ' ')
		(1, 'food') (2, 'drink') (3, 'fruit')

11.8.5 字典的遍历

字典 d 及其视图 d.items()、d.values()、d.keys()都是可迭代对象，可以使用 for 循环进行迭代。例如：

```
>>> d = {1:'food', 2:'drink', 3:'fruit'}
>>> for k in d:
        print("键 = {},值 = {};".format(k, d[k]), end = " ")
键 = 1,值 = food; 键 = 2,值 = drink; 键 = 3,值 = fruit;
>>> for k, v in d.items():
        print("键 = {},值 = {};".format(k, v), end = " ")
键 = 1,值 = food; 键 = 2,值 = drink; 键 = 3,值 = fruit;
>>> for (k, v) in d.items():
        print("键 = {},值 = {};".format(k, v), end = " ")
键 = 1,值 = food; 键 = 2,值 = drink; 键 = 3,值 = fruit;
```

11.8.6　字典解析表达式

使用字典解析表达式可以简单、高效地处理一个可迭代对象,并生成结果字典。字典解析表达式的形式如下:

- **{k:v for i₁ in 序列 1 … for i_N in 序列 N}**　　# 迭代序列中的所有内容,并计算生成字典
- **{k:v for i₁ in 序列 1 … for i_N in 序列 N if cond_expr}**　# 按条件迭代,并计算生成字典

表达式 k 和 v 使用每次迭代内容 $i_1 \sim i_N$ 计算生成一个字典。如果指定了条件表达式 cond_expr,则只有满足条件的元素参与迭代。

【例 11.30】　字典解析表达式示例。

```
>>> {key:value for key in "ABC" for value in range(3)}
{'A': 2, 'B': 2, 'C': 2}
>>> d1 = {1:'food', 2:'drink', 3:'fruit'}
>>> {value:key for key,value in d1.items()}
{'food': 1, 'drink': 2, 'fruit': 3}
>>> {x:x * x for x in range(10) if x % 2  ==  0}
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

11.8.7　判断字典键是否存在

用户可以通过下列方式之一判断键 key 是否存在于字典 d 中:

- **key in d**　　# 如果为 True,表示存在
- **key not in d**　　# 如果为 True,表示不存在

【例 11.31】　判断字典键是否存在示例。

```
>>> d = dict(a = 'apple',b = 'boy',c = 'cat',d = 'dog')
>>> d
{'d': 'dog', 'a': 'apple', 'c': 'cat', 'b': 'boy'}
```

```
>>> 'a' in d
True
>>> 'e' not in d
True
```

11.8.8　字典对象的长度和比较

通过内置函数 len()可以获取字典的长度(元素个数)。虽然字典对象也支持内置函数 max()、min()、sum(),以计算字典 key,但没有太大意义。另外,字典对象还支持比较运算符(<、<=、==、!=、>=、>),但只有==、!=有意义。

【例 11.32】 字典对象的长度和比较示例。

```
>>> d1 = {1:'food', 2:'drink'}
>>> d2 = {1:'food', 2:'drink', 3:'fruit'}
>>> d3 = {1:'food', 2:'drink', 3:'fruit'}
>>> len(d1)
2
```

```
>>> d1  ==  d2
False
>>> d2 != d3
False
```

11.8.9 字典对象的方法

字典是可变对象，其包含的主要方法如表 11-11 所示。假设该表中的示例基于 d={1：'food'，2：'drink'，3：'fruit'}。

表 11-11 字典对象的主要方法

方　法	说　明	示　例
d.clear()	删除所有元素	>>> d.clear()；d　#结果：{}
d.copy()	浅复制字典	>>> d1=d.copy()；id(d)，id(d1) (2487537820800，2487537277976)
d.get(k)	返回键 k 对应的值，如果 key 不存在，返回 None	>>> d.get(1)，d.get(5) ('food'，None)
d.get(k，v)	返回键 k 对应的值，如果 key 不存在，返回 v	>>> d.get(1，'无')，d.get(5，'无') ('food'，'无')
d.pop(k)	如果键 k 存在，返回其值，并删除该项目；否则将导致 KeyError	>>> d.pop(1)，d ('food'，{2：'drink'，3：'fruit'})
d.pop(k，v)	如果键 k 存在，返回其值，并删除该项目；否则返回 v	>>> d.pop(5，'无')，d ('无'，{1：'food'，2：'drink'，3：'fruit'})

d.setdefault(k，v)	如果键 k 存在，返回其值；否则添加项目 k=v，v 默认为 None	>>> d.setdefault(1)　#结果：'food'
		>>> d.setdefault(4)；d
		{1：'food'，2：'drink'，3：'fruit'，4：None}

d.update([other])	使用字典或键值对，更新或添加项目到字典 d	>>> d1={1：'食物'，4：'书籍'}
		>>> d.update(d1)；d
		{1：'食物'，2：'drink'，3：'fruit'，4：'书籍'}

11.8.10 defaultdict 对象

collections.defaultdict(function_factory)用于构建类似 dict 的对象 defaultdict。与 dict 的区别是，在创建 defaultdict 对象时可以使用构造函数参数 function_factory 指定其键/值对中值的类型。

```
defaultdict([function_factory[, … ]])          #构造函数
```

其中，可选参数 function_factory 为字典键/值对中值的类型；其他可选参数同 dict 构造函数。

defaultdict 实现了__missing__(key)，即键不存在时返回值的类型(function_factory)对应的默认值，例如数值为 0、字符串为''、list 为[]等。

【例 11.33】　defaultdict 对象示例。

>>> s = [('r', 1), ('g', 2), ('b', 3)]
>>> dd = defaultdict(int, s)
>>> dd
defaultdict(<class 'int'>, {'r': 1, 'g': 2, 'b': 3})
>>> dd['b']
3
>>> dd['w']　　　#不存在时返回默认值 0
0
	>>> s1 = [('r', 1), ('g', 2), ('b', 3), ('r', 4), ('b', 5)]
	>>> dd1 = defaultdict(list)
	>>> for k, v in s1:
	　　dd1[k].append(v)
	>>> list(dd1.items())
	[('r', [1, 4]), ('g', [2]), ('b', [3, 5])]

11.8.11　OrderedDict 对象

collections.OrderedDict 是 dict 的子类，能够记录字典元素插入的顺序。其构造函数如下：

```
collections.OrderedDict([items])                    #构造函数
```

OrderedDict 对象的元素保持插入的顺序，在更新键的值时不改变顺序；若删除项，然后插入与删除项相同的键/值对，则置于末尾。

除了继承 dict 的方法以外，OrderedDict 对象包括下面两个方法。

- popitem(last=True)：弹出最后一个元素(即 LIFO)；如果 last=False，则弹出第 1 个元素(即 FIFO)。
- move_to_end(key, last=True)：移动键 key 到最后；如果 last=False，则移动到最前面。

注意，两个 OrderedDict 对象的相等比较运算(==)与元素的位置顺序有关。OrderedDict 对象支持反向迭代，使用 reversed()反转列表中的元素。

【例 11.34】　OrderedDict 对象示例。

```
>>> from collections import *
>>> d = {'banana':3,'apple':4,'pear':1,'orange':2}
>>> d.items()          #输出:dict_items([('banana', 3), ('apple', 4), ('pear', 1), ('orange', 2)])
>>> sorted(d.items())  #输出:[('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)]
>>> od = OrderedDict(sorted(d.items()))
>>> od                 #输出:OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
>>> od.popitem()       #输出:('pear', 1)
```

11.8.12　ChainMap 对象

collections.ChainMap 对象用于连接多个 map，其构造函数如下：

```
ChainMap(*maps)
```

ChainMap 对象内部包含多个 map 的列表，maps 属性返回这个列表。第一个 map 为子 map，其他 map 为父 map。在查询时，首先查询第一个 map，如果没有查到，则依次查询其他 map。ChainMap 对象只允许更新第一个 map。

ChainMap 对象 cmap 除了支持字典映射的属性和方法以外，还包含下列属性和方法。

- cmap.maps：属性，返回 cmap 对象内部包含的 map 的列表。
- cmap.parents：属性，返回包含其父 map 的新的 ChainMap 对象，即 ChainMap(*d.maps[1:])。
- cmap.new_child()：方法，返回新的 ChainMap 对象，即 ChainMap({}, *d.maps)。

【例 11.35】 ChainMap 对象示例。

```
>>> from collections import *
>>> m1 = {'a':1, 'b':2}; m2 = {'a':2, 'x':3, 'y':4}; m = ChainMap(m1, m2)
>>> m.maps             #输出:[{'a': 1, 'b': 2}, {'a': 2, 'x': 3, 'y': 4}]
>>> m.parents          #输出:ChainMap({'a': 2, 'x': 3, 'y': 4})
>>> m.new_child()      #输出:ChainMap({}, {'a': 1, 'b': 2}, {'a': 2, 'x': 3, 'y': 4})
>>> m['a']             #查询键'a'的值.输出:1
>>> m['x']             #查询键'x'的值.输出:3
>>> m['a'] = 99        #更新键'a'的值为 99
>>> m['x'] = 10        #更新键'x'的值,因为父 map 不能更新,故实际上是在子map 中插入键/值对
>>> m                  #输出:ChainMap({'a': 99, 'b': 2, 'x': 10}, {'a': 2, 'x': 3, 'y': 4})
```

【例 11.36】 ChainMap 对象应用示例(ChainMap.py),程序中的参数值可以为默认值 map1、环境变量 map2、命令行参数 map3,优先顺序为 map3>map2>map1。使用 ChainMap(map3, map2, map1)可以保证优先使用命令行指定的参数。

```
import os, argparse
from collections import *
defaults = {'color': 'red', 'user': 'guest'}
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}
combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

程序运行结果如下。

```
red
guest
```

11.8.13 Counter 对象

collections.Counter 对象(计数器)用于统计各元素的计数,结果为 map,其构造函数如下:

Counter([iterable-or-mapping])

可选参数为序列或字典 map。

【例 11.37】 创建 Counter 对象示例。

```
>>> from collections import *
>>> c1 = Counter()                          #创建空的 Counter 对象
>>> c2 = Counter('banana')                  #基于序列创建 Counter 对象
>>> c3 = Counter({'red': 4, 'blue': 2})     #基于字典映射创建 Counter 对象
>>> c4 = Counter(cats=4, dogs=8)            #基于命名参数创建 Counter 对象
```

Counter 对象支持字典映射的属性和方法,但查询时如果键不存在将不会报错,而是返回值 0。例如:

```
>>> c = Counter({'red': 4, 'blue': 2})
>>> c['green']                              #输出:0
```

Counter 对象 c 还包含下列属性和方法。

- c.elements():返回元素列表,各元素重复的次数为其计数。

- c.most_common([n])：返回计数值最大的 n 个元素的元组(元素，计数)列表。
- c.subtract([iterable-or-mapping])：元素的计数值减去序列或字典中对应元素的计数。

【例 11.38】 Counter 对象方法示例。

```
>>> from collections import *
>>> c = Counter({'r':3, 'g':2, 'b':1, 'y':4, 'w':3})
>>> c.elements()          #输出:<itertools.chain object at 0x000001D906255908>
>>> list(c.elements())    #输出:['y', 'y', 'y', 'y', 'r', 'r', 'r', 'g', 'g', 'b', 'w', 'w', 'w']
>>> c.most_common(2)      #输出:[('y', 4), ('r', 3)]
>>> c.subtract('red');c   #输出:Counter({'y': 4, 'w': 3, 'r': 2, 'g': 2, 'b': 1, 'd': -1, 'e': -1})
```

【例 11.39】 Counter 对象应用示例(counter.py)：统计文本文件中单词的频率，输出最高频率的 5 个单词。

```
import collections, re
path = r'c:\pythonpa\ch11\counter.py'
with open(path, encoding='utf8') as f:
    words = re.findall(r'\w+', f.read().lower())        #读取文本内容，转换为小写
    c = collections.Counter(words)                       #统计各单词的计数
    print(c.most_common(5))                              #最高计数的 5 个单词
```

程序运行结果如下。

```
[('c', 3), ('counter', 2), ('collections', 2), ('f', 2), ('path', 2)]
```

11.9 collections 模块的其他数据结构

11.9.1 namedtuple 对象

元组(tuple)是常用的数据类型，但只能通过索引访问其元素，如果 tuple 中的元素很多，操作起来会比较麻烦，且容易出错。

使用 namedtuple 的构造函数可以定义一个 tuple 的子类命名元组。namedtuple 对象既可以使用元素名称访问其元素，也可以使用索引访问。其构造函数如下：

```
namedtuple(typename, field_names, verbose=False, rename=False)      #构造函数
```

其中，typename 是返回 tuple 的子类的类名；field_names 是命名元组元素的名称，必须为合法的标识符；当 verbose 为 True 时，在创建命名元组后会打印类定义信息；当 rename 为 True 时，如果 field_names 中包含保留关键字，则自动命名为_1、_2 等。

创建的命名元组的类可以通过_fields 返回其字段属性，例如：

```
somenamedtuple._fields
```

【例 11.40】 namedtuple 对象示例。

```
>>> from collections import *
>>> p = namedtuple('Point', ['x', 'y'])
>>> print(p._fields)               #打印类的字段属性.输出:('x', 'y')
>>> p.x=11; p.y=22
>>> p.x + p.y                      #使用元素名称访问命名元组的元素.输出:33
```

namedtuple 创建的类继承于 tuple，包含 3 个额外的方法。

- somenamedtuple._make(iterable)：从指定序列 iterable 构建命名元组对象。

- somenamedtuple._asdict():把命名元组对象转换为 OrderedDict 对象。
- somenamedtuple._replace(kwargs):创建新的命名元组对象,替换指定字段。

【例 11.41】 namedtuple 对象方法示例。

```
>>> from collections import *
>>> p = namedtuple('Point', ['x', 'y'])
>>> t = [3, 4]
>>> p1 = p._make(t); p1              #输出:Point(x=3, y=4)
>>> p1._asdict()                     #输出:OrderedDict([('x', 3), ('y', 4)])
>>> p1._replace(x=30)                #输出:Point(x=30, y=4)
```

【例 11.42】 namedtuple 对象应用示例(namedtuple.py):读取 CSV 格式的 employees.csv 文件的内容(姓名、年龄、职称、系别和工资)。employees.csv 文件的内容如下:

```
Mary    45    Professor    Chinese     8000
Tom     30    Lecturer     Math        5000
Clinton 50    Professor    Computer    9000
```

namedtuple.py 的内容如下:

```
from collections import *
import csv
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv"))):
    print(emp.name, emp.title)
```

程序运行结果如下。

```
Mary Professor
Tom Lecturer
Clinton Professor
```

11.9.2 UserDict、UserList 和 UserString 对象

UserDict、UserList 和 UserString 分别是 dict、list 和 str 的子类,一般用于创建其派生类。

- **UserDict([initialdata])** #构造函数
- **UserList([list])** #构造函数
- **UserString([sequence])** #构造函数

虽然可以直接基于 dict、list 和 str 创建派生类,但 UserDict、UserList 和 UserString 包括一个属性成员——data,用于存放内容,因此可以更方便实现。

【例 11.43】 UserString 对象示例。

```
>>> from collections import *
>>> us = UserString('abc')
>>> us.data                          #输出:'abc'
```

11.10 应用举例

11.10.1 去除列表中的重复项

用户可以通过构造一个集合来去除列表中的重复项,但结果不能保证原来的顺序。例如:

```
>>> a = [1, 8, 5, 1, 9, 2, 1, 10]
>>> list(set(a))                     #输出:[1, 2, 5, 8, 9, 10]
```

通过定义一个生成器函数可以实现去除列表中重复元素的同时保持原来顺序的功能。

【例 11.44】　去除列表中的重复项生成器函数(deduplicate.py)。

```
def unique(items):
    items_existed = set()
    for item in items:
        if item not in items_existed:
            yield item
            items_existed.add(item)
if __ name__ == "__main__":
    #测试代码
    a = [1, 8, 5, 1, 9, 2, 1, 10]
    a1 = unique(a)
    print(list(a1))
```

程序运行结果如下。

```
[1, 8, 5, 9, 2, 10]
```

11.10.2　基于字典的通讯录

本节实现一个简单的基于字典数据结构的通讯录管理系统,该系统采用 JSON 文件来保存数据。通讯录设计为字典{name:tel}。程序开始时从 addressbook.json 文件中读取通讯录,然后显示主菜单,具体包括如下功能。

(1) 显示通讯录清单:如果通讯录字典中存在用户信息,则显示通讯录清单,包括姓名和电话号码;如果通讯录字典中不存在任何用户信息,则提示"通讯录为空"。

(2) 查询联系人资料:提示用户输入姓名 name,在通讯录字典中查询该键,如果存在,输出联系人信息;如果不存在,提示是否新建联系人。

(3) 插入新的联系人:提示用户输入姓名 name,在通讯录字典中查询该键,如果存在,提示是否更新联系人信息;如果不存在,提示输入电话号码,并插入字典键/值对。

(4) 删除已有联系人:提示用户输入姓名 name,在通讯录字典中查询该键,如果不存在,输出"联系人不存在"的提示信息;如果存在,从通讯录字典中删除键/值对,并输出信息。

(5) 退出:保存通讯录字典到 addressbook.json 中,退出循环。

【例 11.45】　基于字典的通讯录(addressbook.py)。

```
"""简易通讯录程序"""
import os, json
ab = {}                                    #通讯录保存在字典中
#从 JSON 文件中读取通讯录
if os.path.exists("addressbook.json"):
    with open(r'addressbook.json', 'r', encoding = 'utf-8') as f:
        ab = json.load(f)
while True:
    print("|---欢迎使用通讯录程序---|")
    print("|---1:显示通讯录清单 ---|")
    print("|---2:查询联系人资料 ---|")
    print("|---3:插入新的联系人 ---|")
    print("|---4:删除已有联系人 ---|")
    print("|---0:退出 -------------|")
    choice = input('请选择功能菜单(0-3):')
    if choice == '1':
```

```
        if(len(ab) == 0):
            print("通讯录为空")
        else:
            for k, v in ab.items():
                print("姓名 = {},联系电话 = {}".format(k, v))
    elif choice == '2':
        name = input("请输入联系人姓名:")
        if(name not in ab):
            ask = input("联系人不存在,是否增加用户资料(Y/N)")
            if ask in ["Y", "y"]:
                tel = input("请输入用户联系电话:")
                ab[name] = tel
        else:
            print("联系人信息:{} {}".format(name, ab[name]))
    elif choice == '3':
        name = input("请输入联系人姓名:")
        if(name in ab):
            print("已存在联系人:{} {}".format(name, ab[name]))
            ask = input("是否修改用户资料(Y/N)")
            if ask in ["Y", "y"]:
                tel = input("请输入用户联系电话:")
                dict[name] = tel
        else:
            tel = input("请输入用户联系电话:")
            ab[name] = tel
    elif choice == '4':
        name = input("请输入联系人姓名:")
        if(name not in ab):
            print("联系人不存在:{}".format(name))
        else:
            tel = ab.pop(name)
            print("删除联系人:{} {}".format(name, tel))
    elif choice == '0':                       # 保存到 JSON 文件并退出循环
        with open(r'addressbook.json', 'w', encoding = 'utf - 8') as f:
            json.dump(ab, f)
        break
```

11.11 复习题

一、选择题

1. Python 语句 print(type({}))的输出结果是________。
   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>
2. Python 语句 print(type([]))的输出结果是________。
   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>
3. Python 语句 print(type(()))的输出结果是________。
   A. <class 'tuple'>　B. <class 'dict'>　C. <class 'set'>　D. <class 'list'>
4. 以下不能创建字典的 Python 语句是________。
   A. dict1 = {}　B. dict2 = {2:6}
   C. dict3 = {[1,2,3]: "users"}　D. dict4 = {(1,2,3): "users"}

5. 以下不能创建字典的 Python 语句是________。

A. dict1 = {}　　B. dict2 = {1:8}

C. dict3 = dict([2,4],[3,6])　　D. dict4 = dict(([2,4],[3,6]))

二、填空题

1. Python 语句 print(len({}))的输出结果是________。

2. Python 语句序列“d={1: 'x',2: 'y',3: 'z'}; del d[1]; del d[2]; d[1]='A'; print(len(d))”的输出结果是________。

3. Python 语句 print(set([1, 2, 1, 2, 3]))的结果是________。

4. Python 语句“fruits={'apple': 3,'banana': 4,'pear': 5}; fruits['banana']=7; print(sum(fruits.values()))”的结果是________。

5. Python 语句“d1={1: 'food'}; d2={1: '食品',2: '饮料'}; d1.update(d2); print(d1[1])”的结果是________。

6. Python 语句“names = ['Amy', 'Bob', 'Charlie', 'Daling']; print(names[-1][-1])”的结果是________。

7. 在 Python 中设有 s1={1,2,3}、s2={2,3,5},则 s1.update(s2)、s1.intersection_update(s2)、s1.difference_update(s2)、s1.symmetric_difference_update(s2)、s1.add('x')、s1.remove(1)、s1.discard(3)、s1.clear()分别执行后,s1 的结果分别为________。

8. 对于如下 Python 程序代码,其运行时间为________,算法的时间复杂度(增长量级)为________。

```
for i in range(1, n):
    s = ""
    for j in range(1, n):
        s += str.format("{0:1} * {1:1} = {2:<2} ", i, j, i * j)
    print(s)
```

9. 在 Python 中,使用函数________可以返回一个内置数据类型 x 在系统中所占用的字节数。

10. 数据结构通常由 3 个部分组成,即数据的________结构、数据的________结构和数据的________结构。

三、思考题

1. 阅读下面的 Python 语句,请问输出结果是什么?

```
list1 = {};list1[1] = 1;list1['1'] = 3;list1[1] += 2; sum = 0
for k in list1: sum += list1[k]
print(sum)
```

2. 阅读下面的 Python 语句,请问输出结果是什么?

```
d = {1:'a', 2:'b', 3:'c'};
del d[1];d[1] = 'x';del d[2];print(d)
```

3. 阅读下面的 Python 语句,请问输出结果是什么?

```
item_counter = {}
def addone(item):
    if item in item_counter: item_counter[item] += 1
    else: item_counter[item] = 1
addone('Apple');addone('Pear');addone('apple')
```

```
addone('Apple');addone('kiwi');addone('apple')
print(item_counter)
```

4. 阅读下面的 Python 语句，请问输出结果是什么？

```
numbers = {};numbers[(1,2,3)] = 1;
numbers[(2,1)] = 2;numbers[(1,2)] = 3; sum = 0
for k in numbers: sum += numbers[k]
print(len(numbers),' ',sum,' ',numbers)
```

5. 阅读下面的 Python 语句，请问输出结果是什么？

```
d1 = {'a':1, 'b':2};d2 = d1;d1['a'] = 6
sum = d1['a'] + d2['a']
print(sum)
```

6. 阅读下面的 Python 语句，请问输出结果是什么？

```
d1 = {'a':1, 'b':2};d2 = dict(d1);d1['a'] = 6
sum = d1['a'] + d2['a']
print(sum)
```

7. 下列 Python 语句的执行结果是________。

```
from collections import *
m1 = {1:'a', 2:'b'}; m2 = {2:'a', 3:'x', 4:'y'}; m = ChainMap(m1, m2)
print(m.maps,m.parents,m.new_child())
print(m[1],m[3]); m[1] = 'A';m[3] = 'X';print(m)
```

8. 下列 Python 语句的执行结果是________。

```
from collections import *
c1 = Counter();print(c1); c2 = Counter('banana');print(c2)
c3 = Counter({'R': 4, 'B': 2});print(c3)
c4 = Counter(birds = 2, cats = 4, dogs = 8);print(c4)
print(c4['flowers'],c4['cats']); print(list(c3.elements()))
print(c4.most_common(2)); c3.subtract('RGB');print(c3)
```

9. 下列 Python 语句的执行结果是________。

```
from collections import *
dq = deque(); dq.append('a'); dq.append(2); dq.append('c'); data = iter(dq)
while True:
    try: i = next(data)
    except StopIteration: break
    print(i, end = ' ')
print(dq.pop(),dq.pop(),dq.pop())
```

10. 下列 Python 语句的执行结果是________。

```
from collections import *
dq = deque(); dq.append('a'); dq.append(2); dq.append('c')
print(dq.popleft(),dq.popleft(),dq.popleft())
```

11. 下列 Python 语句的执行结果是________。

```
from collections import defaultdict
s = [('r', 3), ('g', 2), ('b', 1)]; dd = defaultdict(int, s); print(dd['b'],dd['w'])
s1 = [('r', 3), ('g', 2), ('b', 1), ('r', 5), ('b', 4)]; dd1 = defaultdict(list)
for k, v in s1: dd1[k].append(v)
print(list(dd1.items()))
```

12. 下列 Python 语句的执行结果是________。

```
from collections import *
d = {'red':3,'green':4,'blue':1}; print(d.items(),sorted(d.items()))
od = OrderedDict(sorted(d.items())); print(od.popitem(),od.popitem(False))
```

13. 下列 Python 语句的执行结果是________。

```
from collections import *
p = namedtuple('Point', ['x', 'y']); p.x=1; p.y=2; print(p._fields,p.x,p.y)
t = [10, 20]; p1 = p._make(t);print(p1._asdict())
print(p1._replace(x=100),p1.x,p1.y)
```

14. 下列 Python 语句的执行结果是________。

```
import array; arr1 = array.array('i', (1, 2, 3, 4, 5))
arr1[1]=22; print(arr1,arr1[2:],type(arr1[1]))
del arr1[2:]; print(arr1,arr1.typecode, arr1.itemsize)
```

15. 下列 Python 语句的执行结果是________。

```
import array; a = array.array('b', (3,2)); a.append(3);a.extend((3,5))
print(a,a.count(3)); a.frombytes(b'A1');a.fromlist([8,9])
print(a,a.index(3));a.insert(0,1);a.pop()
a.remove(2);a.reverse();print(a.tolist())
```

11.12　上机实践

1. 完成本章中的例 11.1～例 11.45，熟悉 Python 语言算法与数据结构的程序设计。

2. 修改例 11.3 在列表中顺序查找特定数值的程序，设法从命令行参数中获取要查询的数据。

3. 修改例 11.4 在列表中顺序查找最大值和最小值的示例程序，设法从命令行参数中获取测试列表的各元素。

4. 修改例 11.5 二分查找法的递归程序，设法从命令行参数中获取测试列表的各元素以及所要查找的关键字。

5. 修改例 11.6 二分查找法的非递归程序，设法从命令行参数中获取测试列表的各元素以及所要查找的关键字。

6. 修改例 11.8 冒泡排序算法程序，设法从命令行参数中获取测试列表的各元素。

7. 修改例 11.9 选择排序算法程序，设法从命令行参数中获取测试列表的各元素。

8. 修改例 11.10 插入排序算法程序，设法从命令行参数中获取测试列表的各元素。

9. 修改例 11.11 归并排序算法程序，设法从命令行参数中获取测试列表的各元素。

10. 修改例 11.12 快速排序算法程序，设法从命令行参数中获取测试列表的各元素。

11. 参照例 11.42 实现 namedtuple 对象应用程序，读取成绩文件 scores.csv 的内容(学员 ID、语文、数学、外语和信息，内容如图 11-3(a)所示)，显示学员 ID 和平均成绩。其运行效果如图 11-3(b)所示。

```
2018111,97,92,81,60
2018112,75,84,91,39
2018113,88,94,65,91
2009114,97,89,85,82
2018115,35,72,91,70
2018116,99,86,90,94
```

(a) 文件scores.csv的内容

学号	平均成绩
2018111	82.5
2018112	72.25
2018113	84.5
2009114	88.25
2018115	67.0
2018116	92.25

(b) 显示学员ID和平均成绩

图 11-3　学生信息运行效果

12. 创建由'Monday'～'Sunday'7 个值组成的字典，输出键列表、值列表以及键值列表。其运行效果如图 11-4 所示。

```
1 2 3 4 5 6 7

Mon Tues Wed Thur Fri Sat Sun

(1, 'Mon') (2, 'Tues') (3, 'Wed') (4, 'Thur') (5, 'Fri') (6, 'Sat') (7, 'Sun')
```

图 11-4　字典运行效果

13. 随机生成 10 个 0（含）～10（含）的整数，分别组成集合 A 和集合 B，输出 A 和 B 的内容、长度、最大值、最小值以及它们的并集、交集和差集。其运行效果如图 11-5 所示。

```
集合的内容、长度、最大值、最小值分别为：
{0, 8, 10, 5, 7} 5 10 0
{9, 2, 10, 5, 6} 5 10 2
A和B的并集、交集和差集分别为：
{0, 2, 5, 6, 7, 8, 9, 10} {10, 5} {0, 8, 7}
```

图 11-5　集合运行效果

11.13　案例研究：程序运行时间度量分析

本章案例研究通过使用 time 模块定义用于程序运行时间度量分析的函数，以显示当求解问题规模 N 增大时给定算法（函数）的时间复杂度增长量级，帮助读者深入了解算法的时间复杂度。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 12 章

图形用户界面



视频讲解

相对于字符界面的控制台应用程序，基于图形化用户界面(Graphic User Interface，GUI)的应用程序可以提供丰富的用户交互界面，从而实现各种复杂功能的应用程序。

12.1　图形用户界面概述

12.1.1　tkinter

tkinter(Tk interface，tk 接口)是 Tk 图形用户界面工具包标准的 Python 接口。tkinter 是 Python 的标准 GUI 库，支持跨平台的图形用户界面应用程序开发，包括 Windows、Linux、UNIX 和 Macintosh 操作系统。

tkinter 的特点是简单、实用。tkinter 是 Python 语言的标准库之一，Python 自带的 IDLE 就是用它开发的。用 tkinter 开发的图形界面，显示风格是本地化的。

tkinter 适用于小型图形界面应用程序的快速开发。本章基于 tkinter 阐述图形用户界面应用程序开发的主要流程。

12.1.2　其他 GUI 库简介

1. pyGtk

Gtk 是 Linux 下 Gnome 的核心 GUI 开发库，功能齐全。pyGtk 模块是 Gnome 图形用户界面工具包标准的 Python 接口。glade 界面设计器支持快速开发 pyGtk 图形界面用户程序。

2. PyQT

Qt 是一种开源的 GUI 库，Qt 的类库大约有 300 多个，函数大约有 5700 多个。Qt 适合于大型应用程序开发。PyQT 模块是 Qt 图形用户界面工具包标准的 Python 接口。Qt Designer 界面设计器支持快速开发 PyQT 图形界面用户程序。

3. wxPython

wxWidgets 是比较流行的 GUI 跨平台开发技术，适合于大型应用程序开发。wxPython 模块是 wxWidgets 图形用户界面工具包标准的 Python 接口，其功能强于 tkinter，设计的框架类似于 MFC(Microsoft Foundation Classes，微软基础类)。Boa Constructor 支持快速开发 wxPython 图形界面用户程序。

4. Jython

Jython 是 Python 的 Java 实现，故可以访问 Java 类库，使用 Java 的 Swing 技术构建图形用户界面程序。

5. IronPython

IronPython 是 Python 的.NET 实现，故可以访问.NET 类库，使用.NET 类库技术构建图形用户界面程序。

12.2 tkinter 概述

12.2.1 tkinter 模块

tkinter 由若干模块组成，例如_tkinter、tkinter 和 tkinter.constants 等。

_tkinter 是二进制扩展模块，提供了对 Tk 的低级接口，应用级程序员不会直接使用。_tkinter 通常是一个共享库（或 DLL），但是在一些情况下也可以被 Python 解释器静态链接。

tkinter 是主要使用的模块，在导入 tkinter 时会自动导入 tkinter.constants。tkinter.constants 模块定义了许多常量。

12.2.2 图形用户界面的构成

基于 tkinter 模块创建的图形用户界面通常包括如下内容。

(1) 通过类 Tk 的无参构造函数创建应用程序主窗口（也称根窗口、顶层窗口）。

```
from tkinter import *                     #导入 tkinter 模块的所有内容
root = Tk()                               #创建一个 Tk 根窗口组件 root
```

(2) 在应用程序主窗口中添加各种可视化组件，例如文本框（Label）、按钮（Button）等。通过对应组件类的构造函数可以创建其实例并设置其属性。例如：

```
btnSayHi = Button(root)                   #创建一个按钮组件 btnSayHi,作为 root 的子组件
btnSayHi["text"] = "Hello"                #设置 btnSayHi 的 text 属性
```

(3) 调用组件的 pack()/grid()/place()方法，通过几何布局管理器（Geometry Manager）调整其显示位置和大小。例如：

```
btnSayHi.pack()                           #调用组件的 pack()方法,调整其显示位置和大小
```

(4) 通过绑定事件处理程序响应用户操作（如单击按钮）引发的事件。例如：

```
def sayHi(e):                                       #定义事件处理程序
    messagebox.showinfo("Message","Hello, world!")  #弹出消息框
btnSayHi.bind("<Button-1>",sayHi)                   #绑定事件处理程序
root.mainloop()                                     #调用组件的 mainloop()方法,进入事件循环
```

【例 12.1】 创建图形用户界面程序（Hello1.py）：创建应用程序主窗口，在应用程序主窗口中单击 Hello 按钮，将弹出“Hello，world!”消息框，程序运行结果如图 12-1 所示。

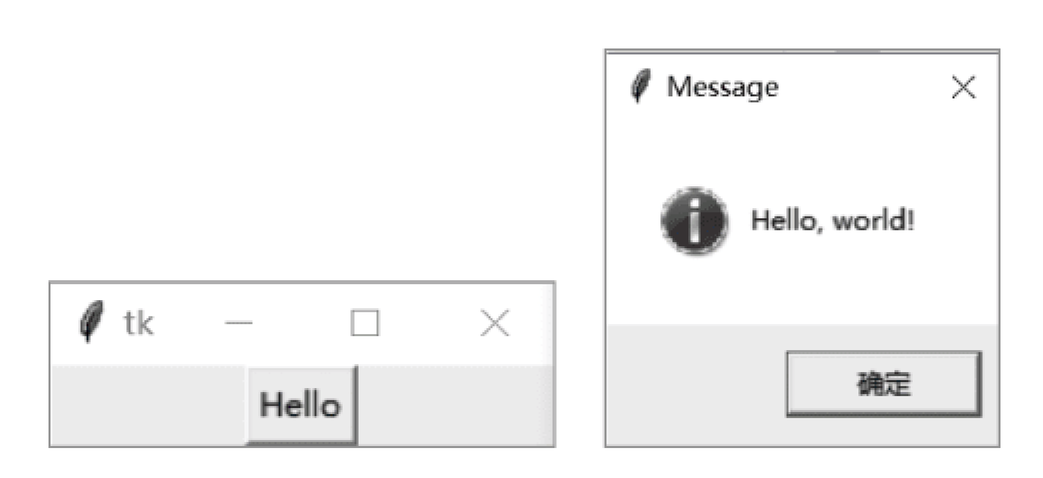


图 12-1 图形用户界面程序

```
from tkinter import *                              # 导入 tkinter 模块的所有内容
from tkinter import messagebox                     # 导入 tkinter 模块中的子模块 messagebox
root = Tk()                                        # 创建一个 Tk 根窗口组件 root
btnSayHi = Button(root)                            # 创建一个按钮组件 btnSayHi,作为 root 的子组件
btnSayHi["text"] = "Hello"                         # 设置 btnSayHi 的 text 属性
btnSayHi.pack()                                    # 调用组件的 pack()方法,调整其显示位置和大小
def sayHi(e):                                      # 定义事件处理程序
  messagebox.showinfo("Message","Hello, world!")   # 弹出消息框
btnSayHi.bind("<Button-1>",sayHi)                  # 绑定事件处理程序
root.mainloop()                                    # 调用组件的 mainloop()方法,进入事件循环
```

12.2.3 框架和 GUI 应用程序类

框架(Frame)是 tkinter 组件之一,表示屏幕上的一块矩形区域。框架一般作为容器使用,在框架中可以包含其他组件,从而实现复杂界面的布局窗体。

在开放正规和复杂的 GUI 应用程序时,一般创建一个继承于 Frame 的类 Application,在其构造函数中调用创建其子组件的方法 createWidgets()。

通过创建 Application 的对象实例可以运行 GUI 应用程序。

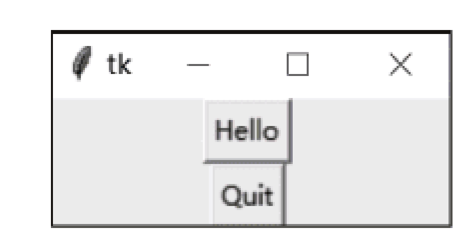


图 12-2 利用框架创建 GUI 应用程序

【例 12.2】 创建 GUI 应用程序类(Hello2.py),实现例 12.1 程序:利用框架创建 GUI 应用程序,在应用程序窗口中分别设计并实现 Hello 按钮和 Quit 按钮响应功能。程序运行结果如图 12-2 所示。

```
import tkinter as tk                               # 导入 tkinter 模块
from tkinter import messagebox                     # 导入 tkinter 模块中的子模块 messagebox
class Application(tk.Frame):                       # 定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master=None):               # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)            # 调用父类的构造函数
        self.pack()                                # 调用组件的 pack()方法,调整其显示位置和大小
        self.createWidgets()                       # 调用对象方法,创建子组件
    def createWidgets(self):                       # 对象方法:创建子组件
        self.btnSayHi = tk.Button(self)            # 创建按钮组件 btnSayHi
        self.btnSayHi["text"] = "Hello"            # 设置显示文本属性
        self.btnSayHi["command"] = self.sayHi      # 设置命令属性,绑定事件处理程序
        self.btnSayHi.pack()                       # 调用组件的 pack()方法,调整其显示位置和大小
        # 创建按钮组件 btnQuit,其显示文本为"Quit",命令事件处理程序为 root.destroy
        self.btnQuit = tk.Button(self, text="Quit", command=root.destroy)
        self.btnQuit.pack()                        # 调用组件的 pack()方法,调整其显示位置和大小
    def sayHi(self):                               # 定义事件处理程序
        tk.messagebox.showinfo("Message","Hello, world!")  # 弹出消息框
root = tk.Tk()                                     # 创建一个 Tk 根窗口组件 root
app = Application(master=root)                     # 创建 Application 的对象实例
app.mainloop()                                     # 调用组件的 mainloop()方法,进入事件循环
```

12.2.4 tkinter 主窗口

1. 主窗口属性

通过类 Tk 的无参构造函数可以创建应用程序主窗口。通过其对象方法 title()可以设置窗口标题;通过字典键可以设置其他属性。通过如下命令可以列举字典键:

```
>>> from tkinter import *
>>> root = Tk(); root.keys()
['bd', 'borderwidth', 'class', 'menu', 'relief', 'screen', 'use', 'background', 'bg', 'colormap', '
container', 'cursor', 'height', 'highlightbackground', 'highlightcolor', 'highlightthickness', 'padx
```

```
', 'pady', 'takefocus', 'visual', 'width']
```

例如:

```
>>> root = Tk()                                # 创建一个 Tk 根窗口组件 root
>>> root.title('示例')                         # 设置窗口标题
>>> root['width'] = 300; root['height'] = 50   # 设置窗口宽度和高度
```

2. 主窗口大小和位置

通过 geometry()函数可以设置主窗口的大小和位置,例如:

```
>>> root.geometry('200x50 - 0 + 0')            # 窗口大小为 200×50,位于屏幕右上角
```

其中,参数的形式为'wxh±x±y'。w 为宽度;h 为高度;+x 为主窗口左边离屏幕左边的距离,-x 为主窗口右边离屏幕右边的距离;+y 为主窗口上边离屏幕上边的距离,-y 为主窗口下边离屏幕下边的距离。

12.3 几何布局管理器

tkinter 几何布局管理器用于组织和管理父组件中子配件的布局方式。tkinter 提供了 3 种不同的几何布局管理类,即 pack、grid 和 place。

12.3.1 pack 几何布局管理器

pack 几何布局管理器采用块的方式组织组件。pack 根据组件创建生成的顺序将子组件添加到父组件中,通过设置选项可以控制子组件的位置等。采用 pack 的代码量最少,故它在快速生成界面设计中被广泛采用。

调用子组件的方法 pack(),则该子组件在其父组件中采用 pack 布局。

```
pack(option = value, … )
```

pack()方法提供了如表 12-1 所示的若干选项。

表 12-1 pack()方法提供的选项

选 项	意 义	取值范围及说明
side	停靠在父组件的哪一边上	'top'(默认值)、'bottom'、'left'、'right'
anchor	停靠的对齐方式。对应于东、南、西、北、中以及 4 个角	'n'、's'、'w'、'e'、'nw'、'sw'、'se'、'ne'、'center'(默认值)
fill	填充空间	'x'、'y'、'both'、'none'
expand	扩展空间	0 或 1
ipadx, ipady	组件内部在 x/y 方向上填充的空间大小	单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点)
padx, pady	组件外部在 x/y 方向上填充的空间大小	同上

【例 12.3】 pack 几何布局示例(pack.py)。程序运行效果如图 12-3 所示。

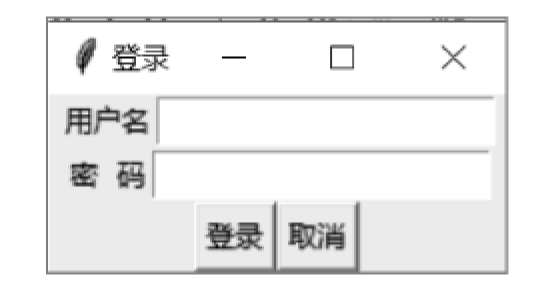


图 12-3 pack 几何布局示例

```
from tkinter import *                                 #导入 tkinter 模块的所有内容
root = Tk(); root.title("登录")                       #窗口标题
f1 = Frame(root); f1.pack()                           #界面分为上、中、下 3 个 Frame,f1 放置第 1 行标签和
                                                      #文本框
f2 = Frame(root); f2.pack()                           #f2 放置第 2 行标签和文本框
f3 = Frame(root); f3.pack()                           #f3 放置第 3 行的两个按钮
Label(f1, text="用户名").pack(side=LEFT)              #标签放置在 f1 中,左停靠
Entry(f1).pack(side=LEFT)                             #单行文本框放置在 f1 中,左停靠
Label(f2, text="密 码").pack(side=LEFT)               #标签放置在 f2 中,左停靠
Entry(f2, show="*").pack(side=LEFT)                   #单行文本框放置在 f2 中,左停靠
Button(f3, text="登录").pack(side=RIGHT)              #按钮放置在 f3 中,右停靠
Button(f3, text="取消").pack(side=RIGHT)              #按钮放置在 f3 中,右停靠
root.mainloop()                                       #调用组件的 mainloop()方法,进入事件循环
```

12.3.2　grid 几何布局管理器

grid 几何布局管理器采用表格结构组织组件。子组件的位置由行/列确定的单元格决定，子组件可以跨越多行/列。在每一列中，列宽由这一列中最宽的单元格确定。grid 适合于表格形式的布局，可以实现复杂的界面，因此被广泛采用。

调用子组件的方法 grid()，则该子组件在其父组件中采用 grid 布局。

```
grid(option=value, …)
```

grid()方法提供了如表 12-2 所示的若干选项。

表 12-2　grid()方法提供的选项

选　　项	意　　义	取值范围及说明
column	单元格列号	从 0 开始的正整数
columnspan	列跨度	正整数
row	单元格行号	从 0 开始的正整数
rowspan	行跨度	正整数
ipadx，ipady	组件内部在 x/y 方向上填充的空间大小	单位为 c(厘米)、m(毫米)、i(英寸)、p(打印机的点)
padx，pady	组件外部在 x/y 方向上填充的空间大小	同上
sticky	组件紧贴所在单元格的某一边角，对应于东、南、西、北、中以及 4 个角	'n'、's'、'w'、'e'、'nw'、'sw'、'se'、'ne'、'center'(默认值)。注意：可以紧贴多个边角。例如 tk. N+tk. S

【例 12.4】　grid 几何布局示例 1(grid1. py)。程序运行效果如图 12-4 所示。

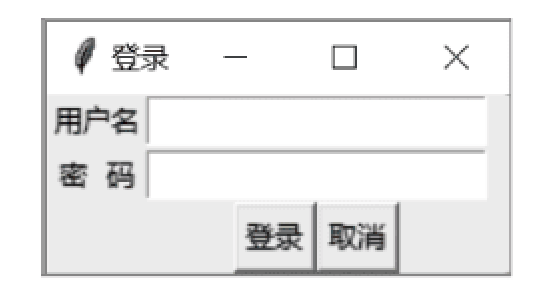


图 12-4　grid 几何布局示例 1

```
from tkinter import *                                 #导入 tkinter 模块的所有内容
root = Tk(); root.title("登录")                       #窗口标题
Label(root, text="用户名").grid(row=0, column=0)      #用户名标签放置在第 0 行第 0 列
```

```
Entry(root).grid(row = 0, column = 1, columnspan = 2)            #用户名文本框放置在第 0 行第
                                                                 #1 列,跨两列
Label(root, text = "密 码").grid(row = 1, column = 0)            #密码标签放置在第 1 行第 0 列
Entry(root, show = " * ").grid(row = 1, column = 1, columnspan = 2) #密码文本框放置在第 1 行第 1
                                                                 #列,跨两列
Button(root, text = "登录").grid(row = 3, column = 1, sticky = E) #"登录"按钮右侧贴紧
Button(root, text = "取消").grid(row = 3, column = 2, sticky = W) #"取消"按钮左侧贴紧
root.mainloop()                                                  #调用组件的 mainloop()方法,
                                                                 #进入事件循环
```

【例 12.5】 grid 几何布局示例 2(grid2.py)。程序运行效果如图 12-5 所示。

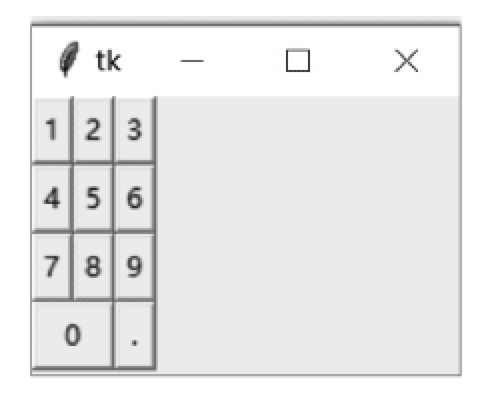


图 12-5 grid 几何布局示例 2

```
from tkinter import *                                        #导入 tkinter 模块的所有内容
root = Tk()
Button(root, text = "1").grid(row = 0, column = 0)           #按钮 1 放置于 0 行 0 列
Button(root, text = "2").grid(row = 0, column = 1)           #按钮 2 放置于 0 行 1 列
Button(root, text = "3").grid(row = 0, column = 2)           #按钮 3 放置于 0 行 2 列
Button(root, text = "4").grid(row = 1, column = 0)           #按钮 4 放置于 1 行 0 列
Button(root, text = "5").grid(row = 1, column = 1)           #按钮 5 放置于 1 行 1 列
Button(root, text = "6").grid(row = 1, column = 2)           #按钮 6 放置于 1 行 2 列
Button(root, text = "7").grid(row = 2, column = 0)           #按钮 7 放置于 2 行 0 列
Button(root, text = "8").grid(row = 2, column = 1)           #按钮 8 放置于 2 行 1 列
Button(root, text = "9").grid(row = 2, column = 2)           #按钮 9 放置于 2 行 2 列
Button(root, text = "0").grid(row = 3, column = 0, columnspan = 2, sticky = E + W)
                                                             #跨两列,左右贴紧
Button(root, text = ".").grid(row = 3, column = 2, sticky = E + W) #左右贴紧
root.mainloop()                                              #调用组件的 mainloop()方法,进
                                                             #入事件循环
```

12.3.3 place 几何布局管理器

place 几何布局管理器允许指定组件的大小与位置。place 的优点是可以精确地控制组件的位置,不足之处是改变窗口大小时子组件不能随之灵活地改变大小。

调用子组件的方法 place(),则该子组件在其父组件中采用 place 布局。

```
place(option, …)
```

place()方法提供了如表 12-3 所示的若干选项,可以直接给选项赋值或对字典变量加以修改。

表 12-3 place()方法提供的选项

选 项	意 义	取值范围及说明
x, y	绝对坐标	从 0 开始的正整数
relx, rely	相对坐标	正整数
width,height	宽和高的绝对值	
relwidth,relheight	宽和高的相对值	
anchor	对齐方式,对应于东、南、西、北、中以及 4 个角	'n'、's'、'w'、'e'、'nw'、'sw'、'se'、'ne'、'center'(默认值)

【例 12.6】　place 几何布局示例(place. py)。程序运行效果如图 12-6 所示。

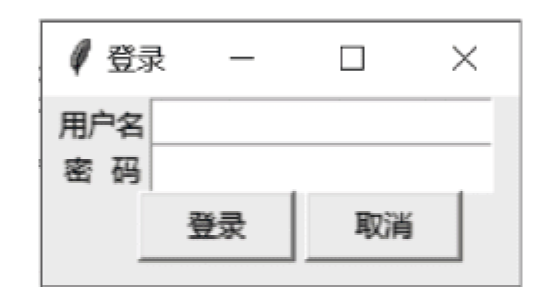


图 12-6　place 几何布局示例

```
from tkinter import *                                           # 导入 tkinter 模块的所有内容
root = Tk();root.title("登录")                                   # 窗口标题
root['width'] = 200; root['height'] = 80                        # 窗口宽度、高度
Label(root, text = "用户名", width = 6).place(x = 1, y = 1)      # 用户名标签,绝对坐标为(1,1)
Entry(root, width = 20).place(x = 45, y = 1)                    # 用户名文本框,绝对坐标为(45,1)
Label(root, text = "密 码",width = 6).place(x = 1, y = 20)       # 密码标签,绝对坐标为(1,20)
Entry(root, width = 20,show = " * ").place(x = 45, y = 20)      # 密码文本框,绝对坐标为(45,20)
Button(root, text = "登录", width = 8).place(x = 40, y = 40)     # "登录"按钮,绝对坐标为(40,40)
Button(root, text = "取消", width = 8).place(x = 110, y = 40)    # "取消"按钮,绝对坐标为(110,40)
root.mainloop()                                                 # 调用组件的 mainloop()方法,进入事
                                                                # 件循环
```

12.4　事件处理

12.4.1　事件类型

用户通过鼠标和键盘与图形用户界面交互时会触发事件。tkinter 事件采用放置于尖括号(<>)内的字符串表示,称之为事件系列(Event sequences)。其通用格式如下:

```
<[modifier - ]…type[ - detail]>
```

其中,可选的 modifier 用于组合键定义,例如同时按下 Ctrl 键;type 表示通用类型,例如键盘按键(KeyPress);可选的 detail 用于具体信息,例如按键 A。

常用的事件类型如下:

```
• < Control - Shift - Alt - KeyPress - A >      # 同时按下 Ctrl、Shift、Alt 和 A 几个键
• < KeyPress - A >                              # 按下键盘上的 A 键
• < Button - 1 >                                # 单击鼠标左键
• < Double - Button - 1 >                       # 双击鼠标左键
```

另外,也可以使用短格式表示事件,例如 '< 1 >' 等同于 '< Button － 1 >','x' 等同于 '< KeyPress－x >'。

12.4.2　事件绑定

1. 在创建组件对象实例时指定

在创建组件对象实例时,可以通过其命名参数 command 指定事件处理函数。

2. 实例绑定

调用组件对象实例方法 bind(),可以为指定组件实例绑定事件,方法如下:

```
w.bind("< event >", eventhandler, add = '')
```

其中,< event >为事件;eventhandler 为事件处理函数;可选参数 add 默认为'',表示事件处理函数代替其他绑定,如果为'+',则加入事件处理队列。

例如绑定组件对象,使得 Canvas 组件实例 canvas1 可以处理鼠标左键事件,代码如下:

```
>>> canvas1 = Canvas(); canvas1.bind("<Button-1>", drawline)
```

3. 类绑定

调用组件对象实例方法 bind_class(),可以为特定组件类绑定事件:

```
w.bind_class("Widget", "<event>", eventhandler, add = '')
```

其中,Widget 为组件类;<event>为事件;eventhandler 为事件处理函数。

例如绑定组件类,使得所有 Canvas 组件实例都可以处理鼠标左键事件:

```
>>> canvas1 = Canvas(); canvas1.bind_class("Canvas", "<Button-1>", drawline)
```

4. 程序界面绑定

调用组件对象实例方法 bind_all(),可以为所有组件类绑定事件:

```
w.bind_all("<event>", eventhandler, add = '')
```

其中,<event>为事件;eventhandler 为事件处理函数。

例如将 PrintScreen 键与程序中的所有组件对象绑定,使得整个程序界面都能处理打印屏幕的键盘事件:

```
>>> canvas1 = Canvas(); canvas1.bind_all("<Key-Print>", printscreen)
```

12.4.3 事件处理函数

1. 定义事件函数和事件方法

事件处理可以定义为函数,也可以定义为对象的方法,两者都带一个参数 event。在触发事件调用事件处理函数时将传递 Event 对象实例。

```
def handlerName(event):
    函数体
def handlerName(self, event):
    方法体
```

2. Event 事件对象参数属性

通过传递的 Event 事件对象的属性可以获取各种相关参数。

【例 12.7】 事件处理示例(event.py):单击鼠标左键,输出坐标位置信息。

```
from tkinter import *                        #导入 tkinter 模块的所有内容
root = Tk();root.title("事件处理")             #窗口标题
def printEvent(event):                       #事件处理函数
    print('当前坐标位置:',event.x, event.y)
root.bind('<Button-1>',printEvent)           #单击鼠标左键
root.mainloop()                              #调用组件的 mainloop()方法,进入事件循环
```

12.5 常用组件

12.5.1 Label

Label(标签)主要用于显示文本信息。Label 既可以显示文本,也可以显示图像。

【例 12.8】 Label 示例(label.py)。

```
from tkinter import *                        #导入 tkinter 模块的所有内容
```

```
root = Tk();root.title("Label 示例")              # 窗口标题
w = Label(root, text = "姓名")                    # 创建 Label 组件对象,显示文本为"姓名"
w.config(width = 20, bg = 'black', fg = 'white')  # 设置宽度、背景色、前景色
w['anchor'] = E                                   # 设置停靠方式为右对齐
w.pack()                                          # 调用 pack()方法,调整其显示位置和大小
root.mainloop()                                   # 调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-7 所示。



图 12-7　Label 示例

12.5.2　LabelFrame

LabelFrame(标签框架)是一个带标签的矩形框架,主要用于包含若干组件。

【例 12.9】　LabelFrame 示例(labelFrame.py)。

```
from tkinter import *                             # 导入 tkinter 模块的所有内容
root = Tk(); root.title("LabelFrame")             # 创建一个 Tk 根窗口组件; 设置 root 窗口标题
lf = LabelFrame(root, text = "组 1")              # 创建 LabelFrame 组件对象
lf.pack()                                         # 调用 pack()方法,调整其显示位置和大小
Button(lf, text = "确定").pack(side = LEFT)       # "确定"按钮,左停靠
Button(lf, text = "取消").pack(side = LEFT)       # "取消"按钮,左停靠
root.mainloop()                                   # 调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-8 所示。



图 12-8　LabelFrame 示例

12.5.3　Button

Button 用于执行用户的单击操作。如果焦点位于某个 Button,则使用鼠标或空格键单击该按钮时会产生 command 事件。

【例 12.10】　Button 示例(button.py)。

```
from tkinter import *                             # 导入 tkinter 模块的所有内容
root = Tk(); root.title("Button")                 # 窗口标题
w = Button(root, text = "确定")                   # 创建 Button 组件对象,显示文本为"确定"
w.config(state = DISABLED)                        # 设置 Button 组件的状态为禁用
w['width'] = 20                                   # 设置宽度
w.pack()                                          # 调用 pack()方法,调整其显示位置和大小
root.mainloop()                                   # 调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-9 所示。



图 12-9　Button 示例

【例 12.11】 Label 和 Button 应用示例（PictureViewer.py）：简易图片浏览器。程序运行效果如图 12-10 所示。



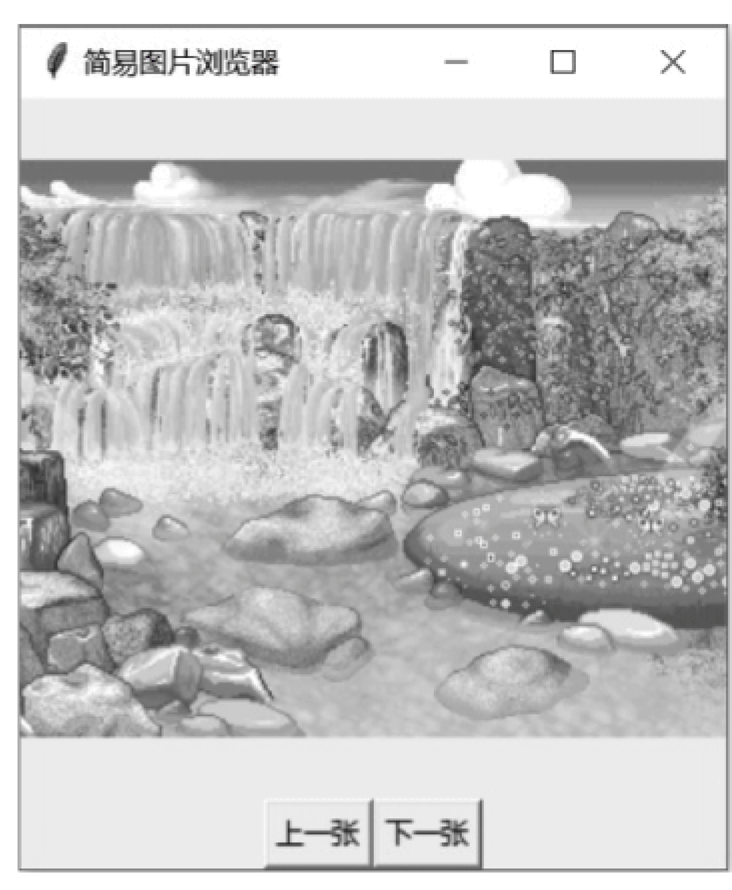


图 12-10 简易图片浏览器程序运行效果

```
import tkinter as tk, os                                  # 导入 tkinter 模块
class Application(tk.Frame):                              # 定义 GUI 应用程序类，派生于 Frame 类
  def __init__(self, master = None):                      # 构造函数，master 为父窗口
    self.files = os.listdir(r'c:\pythonpa\images\gif')
                                                          # 获取图像文件名列表
    self.index = 0                                        # 图片索引，初始显示第一张图片
    self.img = tk.PhotoImage(file = r'c:\pythonpa\images\gif' + '\\' + self.files[self.index])
    tk.Frame.__init__(self, master)                       # 调用父类的构造函数
    self.pack()                                           # 调用组件的 pack()方法，调整其显示位置
                                                          # 和大小
    self.createWidgets()                                  # 调用对象方法，创建子组件
  def createWidgets(self):                                # 对象方法：创建子组件
    self.lblImage = tk.Label(self, width = 300, height = 300)
                                                          # 创建 Label 组件，显示图片
    self.lblImage['image'] = self.img                     # 显示第一张图片
    self.lblImage.pack()                                  # 调用组件的 pack()方法，调整其显示位置
                                                          # 和大小
    self.f = tk.Frame()                                   # 创建窗口框架
    self.f.pack()                                         # 调用组件的 pack()方法，调整其显示位置
                                                          # 和大小
    self.btnPrev = tk.Button(self.f, text = '上一张', command = self.prev)   # 创建按钮组件
    self.btnPrev.pack(side = tk.LEFT)
    self.btnNext = tk.Button(self.f, text = '下一张', command = self.next)   # 创建按钮组件
    self.btnNext.pack(side = tk.LEFT)
  def prev(self):                                         # 定义事件处理程序
    self.showfile(-1)                                     # 显示上一张图片
  def next(self):                                         # 定义事件处理程序
    self.showfile(1)                                      # 显示下一张图片
  def showfile(self, n):                                  # 显示图片
    self.index += n
    if self.index < 0: self.index = len(self.files) - 1  # 循环显示最后一张
    if self.index > len(self.files) - 1: self.index = 0  # 循环显示第一张
    self.img = tk.PhotoImage(file = r'c:\pythonpa\images\gif' + '\\' + self.files[self.index])
    self.lblImage['image'] = self.img
root = tk.Tk()                                            # 创建一个 Tk 根窗口组件 root
```

```
root.title('简易图片浏览器')                      #设置窗口标题
app = Application(master = root)                   #创建 Application 的对象实例
app.mainloop()                                     #调用组件的 mainloop()方法,进入事件循环
```

12.5.4　Message

Message(消息)和 Label 一样,也是用来显示文本信息,但主要用来显示多行文本信息。

【例 12.12】　Message 示例(message.py)。

```
from tkinter import *                              #导入 tkinter 模块的所有内容
root = Tk(); root.title("Message")                 #窗口标题
w = Message(root, bg = 'black', fg = 'white')      #创建 Message 组件对象
w.config(text = "内容显示在一个宽高比为 150 % 的消息框中")
                                                   #设置显示文本
w['anchor'] = W                                    #设置停靠方式为左对齐
w.pack()                                           #调用 pack()方法,调整其显示位置和大小
root.mainloop()                                    #调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-11 所示。

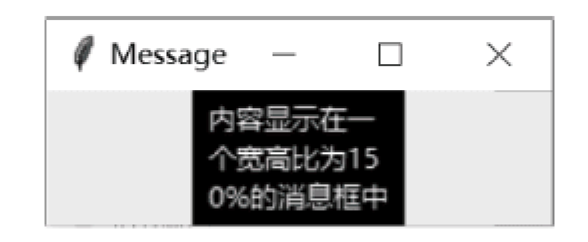


图 12-11　Message 示例

12.5.5　Entry

Entry(单行文本框)主要用于显示和编辑文本。

【例 12.13】　Entry 示例(entry.py)。

```
from tkinter import *                              #导入 tkinter 模块的所有内容
root = Tk(); root.title("Entry")                   #窗口标题
v = StringVar()                                    #创建 StringVar 对象
w1 = Entry(root, textvariable = v)                 #创建 Entry 组件对象
w1.pack()                                          #显示单行文本框
w1.get()                                           #获取组件的内容
v.set('1234')                                      #设置 StringVar 对象的值,组件文本自动更新
root.mainloop()                                    #调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-12 所示。



图 12-12　Entry 示例

12.5.6　Text

Text(多行文本框)主要用于显示和编辑多行文本。

【例 12.14】　Text 示例(text.py)。

```
from tkinter import *                              #导入 tkinter 模块的所有内容
root = Tk(); root.title("Text")                    #窗口标题
```

```
w = Text(root, width = 20, height = 5)           # 创建文本框,宽 20、高 5
w.pack()                                         # 调用 pack()方法,调整其显示位置和大小
w.insert(1.0, '生,还是死,这是一个问题!\n ')
w.get(1.0)                                       # '生'
w.get(1.0, END)                                  # '生,还是死,这是一个问题!\n'
root.mainloop()                                  # 调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-13 所示。

【例 12.15】 Entry 和 Text 应用示例(register.py):用户注册。程序运行效果如图 12-14 所示。

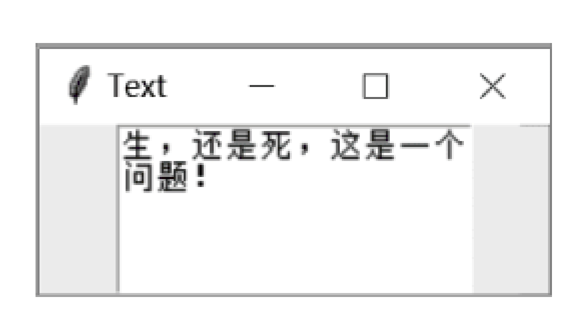
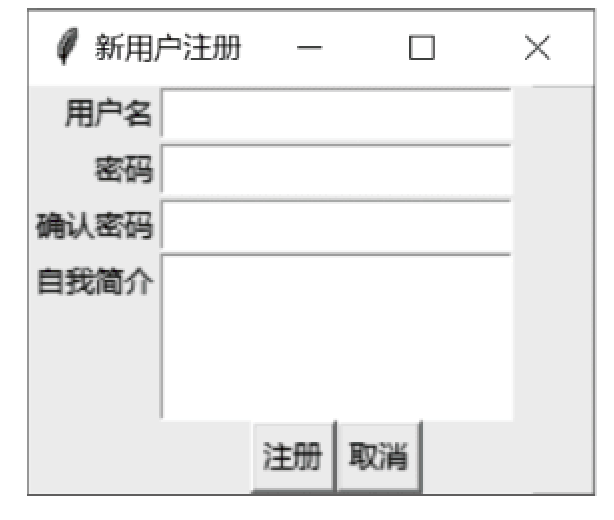


图 12-13 Text 示例



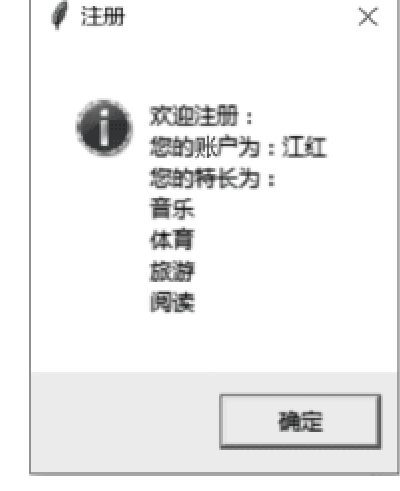


图 12-14 Entry 和 Text 程序的运行效果

```
import tkinter as tk                                          # 导入 tkinter 模块
from tkinter import messagebox                                # 导入 tkinter 模块中的子模块
                                                              # messagebox
class Application(tk.Frame):                                  # 定义 GUI 应用程序类,派生于
                                                              # Frame 类
    def __init__(self, master = None):                        # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                       # 调用父类的构造函数
        self.grid()                                           # 调用组件的 grid()方法,
                                                              # 调整其显示位置和大小
        self.createWidgets()                                  # 调用对象方法,创建子组件
    def createWidgets(self):                                  # 对象方法:创建子组件
        self.lblEmail = tk.Label(self, text = '用户名')        # 创建 Label 组件 - 用户名
        self.lblPass1 = tk.Label(self, text = '密码')          # 创建 Label 组件 - 密码
        self.lblPass2 = tk.Label(self, text = '确认密码')      # 创建 Label 组件 - 确认密码
        self.lblDesc = tk.Label(self, text = '自我简介')       # 创建 Label 组件 - 自我简介
        self.lblEmail.grid(row = 0, column = 0, sticky = tk.E)   # Email 标签放置于 0 行 0 列
        self.lblPass1.grid(row = 1, column = 0, sticky = tk.E)   # 密码标签放置于 1 行 0 列
        self.lblPass2.grid(row = 2, column = 0, sticky = tk.E)   # 确认密码标签放置于 2 行 0 列
        self.lblDesc.grid(row = 3, column = 0, sticky = tk.NE)   # 自我简介标签放置于 3 行 0 列
        self.entryEmail = tk.Entry(self)                      # 创建 Entry 组件
        self.entryPass1 = tk.Entry(self, show = '*')          # 密码默认显示为 *
        self.entryPass2 = tk.Entry(self, show = '*')          # 确认密码默认显示为 *
        self.textDesc = tk.Text(self, width = 20, height = 5)   # 创建 Text 组件
        self.entryEmail.grid(row = 0, column = 1, columnspan = 2) # 用户名文本框放置于 0 行 1 列
        self.entryPass1.grid(row = 1, column = 1, columnspan = 2) # 密码文本框放置于 1 行 1 列
        self.entryPass2.grid(row = 2, column = 1, columnspan = 2) # 确认密码文本框放置于 2 行 1 列
        self.textDesc.grid(row = 3, column = 1, columnspan = 2)   # 自我简介文本框放置于 3 行 1 列
        self.btnOk = tk.Button(self, text = '注册', command = self.funcOK)
                                                              # 创建按钮组件
        self.btnOk.grid(row = 4, column = 1, sticky = tk.E)
                                                              # "注册"按钮放置于 4 行 1 列
        self.btnCancel = tk.Button(self, text = '取消', command = root.destroy)
                                                              # 创建按钮组件
        self.btnCancel.grid(row = 4, column = 2, sticky = tk.W)   # "取消"按钮放置于 4 行 2 列
    def funcOK(self):                                         # 定义注册事件处理程序
```

```
        str1 = '欢迎注册:\n'
        str1 += "您的账户为:" + self.entryEmail.get() + '\n' #获取用户名
        str1 += "您的特长为:\n" + self.textDesc.get(0.0, tk.END)   #获取自我简介
        tk.messagebox.showinfo("注册", str1)                  #弹出消息框
root = tk.Tk()                                               #创建一个 Tk 根窗口组件 root
root.title('新用户注册')                                      #设置窗口标题
app = Application(master=root)                               #创建 Application 的对象实例
app.mainloop()                                               #调用组件的 mainloop()方法,
                                                             #进入事件循环
```

12.5.7　Radiobutton

Radiobutton(单选按钮)控件用于选择同一组单选按钮中的一个单选按钮(不能同时选择多个)。Radiobutton 可以显示文本,也可以显示图像。

【例 12.16】　Radiobutton 示例(radiobutton.py)。

```
from tkinter import *                                 #导入 tkinter 模块的所有内容
root = Tk(); root.title("Radiobutton")                #窗口标题
v = StringVar();v.set('M')                            #创建 StringVar 对象,并设置初始值
w1 = Radiobutton(root, text="男", value='M', variable=v)
w2 = Radiobutton(root, text="女", value='F', variable=v)
w1.pack(side=LEFT)                                    #调用 pack()方法,调整其显示位置
w2.pack(side=LEFT)                                    #调用 pack()方法,调整其显示位置
v.get()                                               #选择女后,获取其值'F'
root.mainloop()                                       #调用组件的 mainloop()方法,进入事件循环
```

程序运行结果如图 12-15 所示。

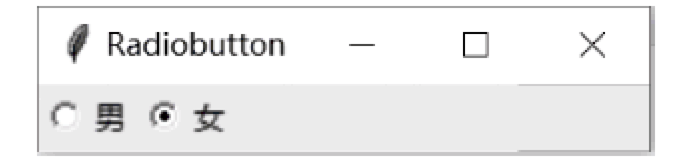


图 12-15　Radiobutton 示例

12.5.8　Checkbutton

Checkbutton(复选框)控件用于选择一个或多个选项(可以同时选择多个)。Checkbutton 可显示文本,也可显示图像。

【例 12.17】　Checkbutton 示例(checkbutton.py)。

```
from tkinter import *                                 #导入 tkinter 模块的所有内容
root = Tk(); root.title("Checkbutton")                #窗口标题
v = StringVar()                                       #创建 StringVar 对象
v.set('yes')                                          #设置默认值为'yes',对应选择状态
w = Checkbutton(root, text="音乐", variable=v, onvalue='yes', offvalue='no')
w.pack()                                              #调用 pack()方法,调整其显示位置和大小
v.get()                                               #用户去选后,获取其值为'no'
root.mainloop()                                       #调用组件的 mainloop()方法,进入事件循环
```

程序运行结果如图 12-16 所示。



图 12-16　Checkbutton 示例

【例 12.18】 Radiobutton 和 Checkbutton 应用示例(Questionnaire.py):实现 Questionnaire 调查个人信息。程序运行效果如图 12-17 所示。

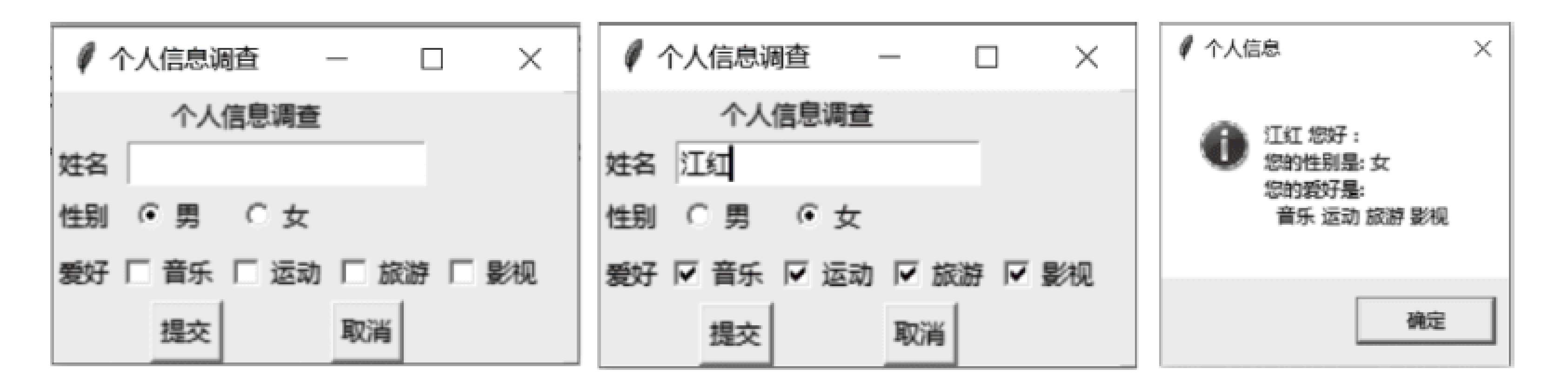


图 12-17 Radiobutton 和 Checkbutton 程序运行效果

```
import tkinter as tk                                    # 导入 tkinter 模块
from tkinter import messagebox                          # 导入 tkinter 模块中的子模块
                                                        # messagebox
class Application(tk.Frame):                            # 定义 GUI 应用程序类,派生于
                                                        # Frame 类
    def __init__(self, master=None):                    # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                 # 调用父类的构造函数
        self.grid()                                     # 调用组件的 grid()方法,调整其显
                                                        # 示位置和大小
        self.createWidgets()                            # 调用对象方法,创建子组件
    def createWidgets(self):                            # 对象方法:创建子组件
        self.lblTitle = tk.Label(self, text='个人信息调查') # 个人信息调查标签
        self.lblName = tk.Label(self, text='姓名')      # 姓名标签
        self.lblSex = tk.Label(self, text='性别')       # 性别标签
        self.lblHobby = tk.Label(self, text='爱好')     # 爱好标签
        self.lblTitle.grid(row=0, column=0, columnspan=4) # 个人信息标签置于 0 行 0 列,跨
                                                          # 4 列
        self.lblName.grid(row=1, column=0)              # 姓名标签置于 1 行 0 列
        self.lblSex.grid(row=2, column=0)               # 性别标签置于 2 行 0 列
        self.lblHobby.grid(row=3, column=0)             # 爱好标签置于 3 行 0 列
        # 文本框
        self.entryName = tk.Entry(self)                 # 创建 Entry 组件"姓名"
        self.entryName.grid(row=1, column=1, columnspan=3) # 姓名文本框置于 1 行 1 列
        # 单选按钮
        self.vSex = tk.StringVar()                      # 创建 StringVar 对象"性别"
        self.vSex.set('M')                              # 设置初始值为男
        self.radioSexM = tk.Radiobutton(self, text="男", value='M', variable=self.vSex)
        # 单选按钮
        self.radioSexF = tk.Radiobutton(self, text="女", value='F', variable=self.vSex)
        self.radioSexM.grid(row=2, column=1)            # "男"单选按钮置于 2 行 1 列
        self.radioSexF.grid(row=2, column=2)            # "女"单选按钮置于 2 行 2 列
        # 复选框
        self.vHobbyMusic = tk.IntVar()                  # 创建 IntVar 对象"音乐"
        self.vHobbySports = tk.IntVar()                 # 创建 IntVar 对象"运动"
        self.vHobbyTravel = tk.IntVar()                 # 创建 IntVar 对象"旅游"
        self.vHobbyMovie = tk.IntVar()                  # 创建 IntVar 对象"影视"
        self.checkboxMusic = tk.Checkbutton(self, text="音乐", variable=self.vHobbyMusic)
                                                        # 音乐
        self.checkboxSports = tk.Checkbutton(self, text="运动", variable=self.vHobbySports)
                                                        # 运动
        self.checkboxTravel = tk.Checkbutton(self, text="旅游", variable=self.vHobbyTravel)
                                                        # 旅游
        self.checkboxMovie = tk.Checkbutton(self, text="影视", variable=self.vHobbyMovie)
                                                        # 影视
        self.checkboxMusic.grid(row=3, column=1)        # "音乐"复选框置于 3 行 1 列
        self.checkboxSports.grid(row=3, column=2)       # "运动"复选框置于 3 行 2 列
        self.checkboxTravel.grid(row=3, column=3)       # "旅游"复选框置于 3 行 3 列
```

```
        self.checkboxMovie.grid(row = 3, column = 4)          #"影视"复选框置于 3 行 4 列
        #按钮
        self.btnOk = tk.Button(self, text = '提交', command = self.funcOK)
                                                              #创建"提交"按钮组件
        self.btnOk.grid(row = 4, column = 1, sticky = tk.E)
                                                              #"提交"按钮置于 4 行 1 列
        self.btnCancel = tk.Button(self, text = '取消', command = root.destroy)
                                                              #创建"取消"按钮组件
        self.btnCancel.grid(row = 4, column = 3, sticky = tk.W)  #"取消"按钮置于 4 行 3 列
    def funcOK(self):                                         #定义提交事件处理程序
        strSex = '男' if (self.vSex.get() == 'M') else '女'
        strMusic = self.checkboxMusic['text'] if (self.vHobbyMusic.get() == 1) else ''
        strSports = self.checkboxSports['text'] if (self.vHobbySports.get() == 1) else ''
        strTravel = self.checkboxTravel['text'] if (self.vHobbyTravel.get() == 1) else ''
        strMovie = self.checkboxMovie['text'] if (self.vHobbyMovie.get() == 1) else ''
        str1 = self.entryName.get() + '您好:\n'
        str1 += "您的性别是: " + strSex + '\n'
        str1 += '您的爱好是:\n ' + strMusic + ' ' + strSports + ' ' + strTravel + ' '
+ strMovie
        tk.messagebox.showinfo("个人信息", str1)               #弹出消息框
root = tk.Tk()                                                #创建一个 Tk 根窗口组件 root
root.title('个人信息调查')                                    #设置窗口标题
app = Application(master = root)                              #创建 Application 的对象实例
app.mainloop()                                                #调用组件的 mainloop0()方法,进
                                                              #入事件循环
```

12.5.9 Listbox

Listbox(列表框)用于显示对象列表,并且允许用户选择一项或多项。

【例 12.19】 Listbox 示例 1(Listbox1.py)。

```
from tkinter import *                          #导入 tkinter 模块的所有内容
root = Tk(); root.title("Listbox")             #窗口标题
v = StringVar()
v.set(('linux','windows','unix'))
lb = Listbox(root, selectmode = EXTENDED, listvariable = v)
lb.pack()                                      #调用 pack()方法,调整其显示位置和大小
for item in ['python','tkinter','widget']: lb.insert(END,item)
#列表框
lb.curselection()                              #选择项目的索引位置:('2', '3')
for i in lb.curselection():print(lb.get(i), end = ' ')
                                               #输出选择项目:unix python
root.mainloop()                                #调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-18 所示。

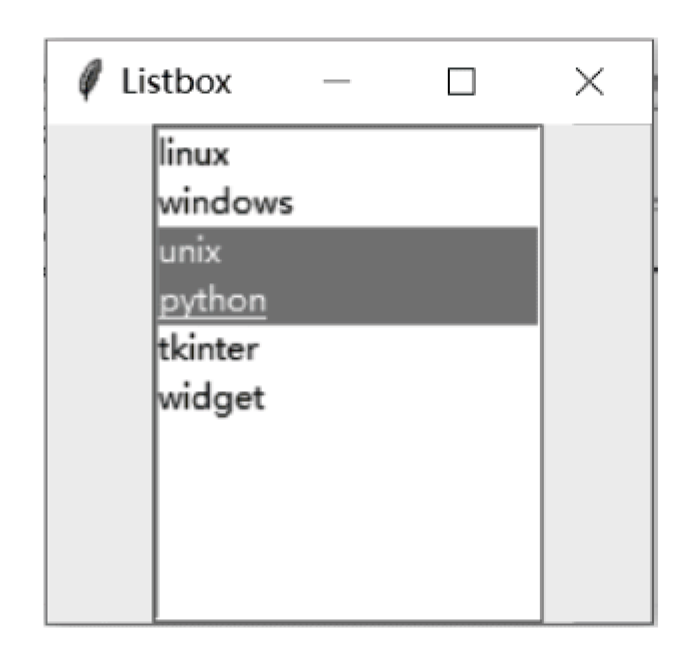


图 12-18 Listbox 示例

【例 12.20】 Listbox 示例 2(Listbox2.py):实现列表选择功能。程序运行效果如图 12-19 所示。



图 12-19 列表选择程序运行效果

```
import tkinter as tk                                      #导入 tkinter 模块
class Application(tk.Frame):                              #定义 GUI 应用程序类,派生于
                                                          #Frame 类
    def __init__(self, master=None):                      #构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                   #调用父类的构造函数
        self.grid()                                       #调用组件的 grid()方法,调整其显示位
                                                          #置和大小
        self.createWidgets()                              #调用对象方法,创建子组件
    def createWidgets(self):                              #对象方法:创建子组件
        self.listboxLeft = tk.Listbox(self, width=10, height=6)   #创建 Listbox 组件
        self.listboxLeft.insert(0, '北京', '天津', '上海', '重庆')   #插入列表数据
        self.listboxLeft.grid(row=0, column=0, rowspan=5)         #置于 0 行 0 列,跨 5 行
        self.listboxRight = tk.Listbox(self, width=10, height=6)  #创建 Listbox 组件
        self.listboxRight.grid(row=0, column=2, rowspan=5)        #置于 0 行 2 列,跨 5 行
        #按钮
        self.btnToRight = tk.Button(self, text='>', command=self.funcToRight)
                                                                  #创建按钮组件
        self.btnToRight.grid(row=1, column=1)                     #置于 1 行 1 列
        self.btnToLeft = tk.Button(self, text='<', command=self.funcToLeft)
                                                                  #创建按钮组件
        self.btnToLeft.grid(row=3, column=1)                      #置于 3 行 1 列
    def funcToRight(self):                                #定义事件处理程序:在右边列表框中显示
                                                          #左边列表框选中的内容
        for item in self.listboxLeft.curselection():      #选中的内容
            self.listboxRight.insert(tk.END, self.listboxLeft.get(item))
                                                          #插入到右边列表框
        for item in self.listboxLeft.curselection():
            self.listboxLeft.delete(item)                 #从左边列表框——删除选中的内容
    def funcToLeft(self):                                 #定义事件处理程序:在左边列表框中显示
                                                          #右边列表框选中的内容
        for item in self.listboxRight.curselection():     #选中的内容
            self.listboxLeft.insert(tk.END, self.listboxRight.get(item))
                                                          #插入左边列表框
        for item in self.listboxRight.curselection():
            self.listboxRight.delete(item)                #从右边列表框——删除选中的内容
root = tk.Tk()                                            #创建一个 Tk 根窗口组件 root
root.title('列表框')                                       #设置窗口标题
app = Application(master=root)                            #创建 Application 的对象实例
app.mainloop()                                            #调用组件的 mainloop()方法,进入事件
                                                          #循环
```

12.5.10 OptionMenu

OptionMenu(选择项)是允许用户选择一项的列表框(在用户请求时显示)。用户单击下拉按钮可以显示列表框,选择的内容会显示在顶部文本框中。

【例 12.21】 OptionMenu 示例 1(OptionMenu1.py)。

```
from tkinter import *                                    # 导入 tkinter 模块的所有内容
root = Tk(); root.title("选择项")                        # 窗口标题
v = StringVar(root)
v.set('Python')
om = OptionMenu(root,v,'Python','Perl','JavaScript','C  #')
om['width'] = 10                                         # 宽度为 10
om['anchor'] = W                                         # 设置停靠对齐方式
om.pack()                                                # 调用 pack()方法,调整其显示位置和大小
root.mainloop()                                          # 调用组件的 mainloop()方法,进入事件循环
```

程序运行效果如图 12-20 所示。

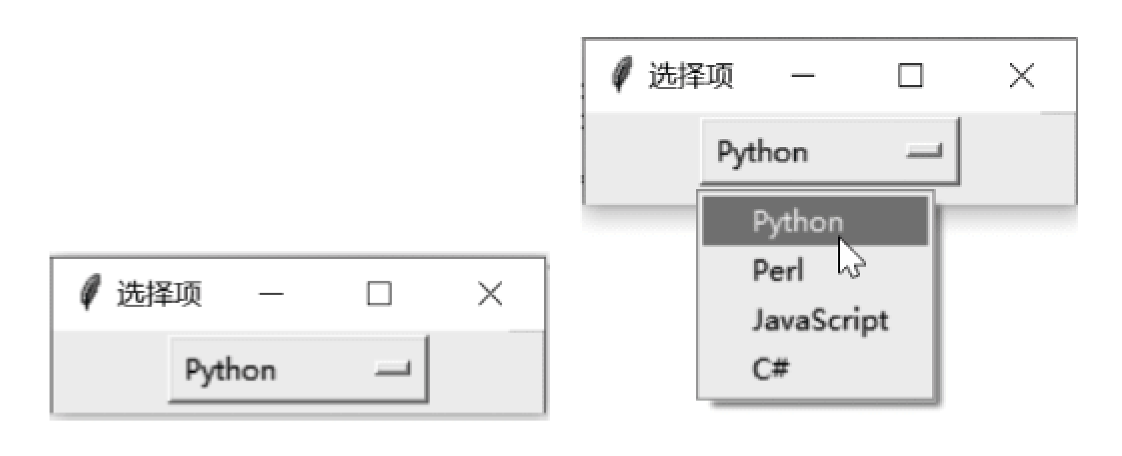


图 12-20 OptionMenu 示例

【例 12.22】 OptionMenu 示例 2(OptionMenu2.py):从选择项中选择字体大小,然后单击“改变字体”按钮改变标签文本的字体大小。程序运行效果如图 12-21 所示。

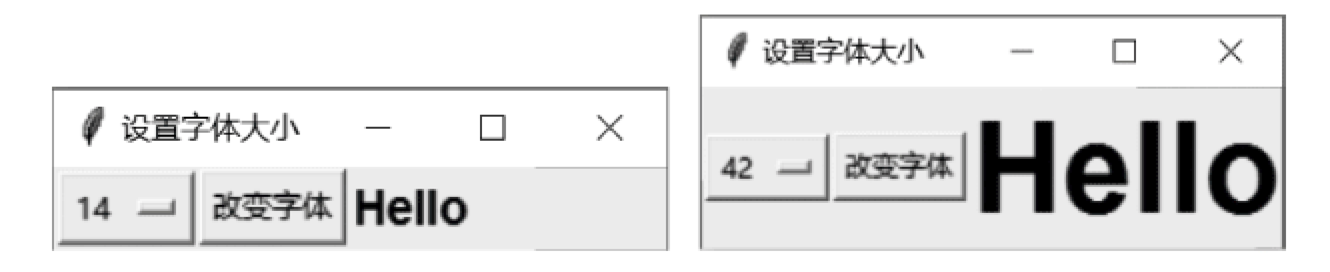


图 12-21 利用 OptionMenu 改变文本字体大小

```
import tkinter as tk                                     # 导入 tkinter 模块
class Application(tk.Frame):                             # 定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master = None):                   # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                  # 调用父类的构造函数
        self.grid()                                      # 调用组件的 grid()方法,调整其显示位置和
                                                         # 大小
        self.createWidgets()                             # 调用对象方法,创建子组件
    def createWidgets(self):                             # 对象方法:创建子组件
        # 创建 Scale 组件
        optionList = range(10,61,4)
        self.vFont = tk.StringVar()
        self.vFont.set(14)                                           # 设置初始值
        self.optionMenuFont = tk.OptionMenu(self, self.vFont, * optionList)
                                                                     # 创建 OptionMenu 组件
        self.optionMenuFont.pack(side = tk.LEFT)                     # 设置停靠对齐方式
        self.buttonFont = tk.Button(self, text = '改变字体', command = self.changefont)
                                                                     # 创建 Button 组件
        self.buttonFont.pack(side = tk.LEFT)                         # 设置停靠对齐方式
        self.lblTitle = tk.Label(self, text = 'Hello', font = ('Helvetica', 14, 'bold'))
                                                                     # 创建 Label 组件
        self.lblTitle.pack(side = tk.LEFT)                           # 设置停靠对齐方式
    def changefont(self):                               # 定义事件处理程序:改变字体
```

```
            fontNew = ('Helvetica', self.vFont.get(), 'bold')
            self.lblTitle.config(font = fontNew)
root = tk.Tk()                                  # 创建一个 Tk 根窗口组件 root
root.title('设置字体大小')                       # 设置窗口标题
root['width'] = 400; root['height'] = 50        # 设置窗口的宽、高
app = Application(master = root)                # 创建 Application 的对象实例
app.mainloop()                                  # 调用组件的 mainloop()方法,进入事件循环
```

12.5.11 Scale

Scale(移动滑块)控件用于在有界区间内通过移动滑块来选择值。

【例 12.23】 Scale 示例(Scale.py)：移动滑块，改变字体大小。程序运行效果如图 12-22 所示。



图 12-22 Scale 程序运行效果

```
import tkinter as tk                            # 导入 tkinter 模块
class Application(tk.Frame):                    # 定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master = None):          # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)         # 调用父类的构造函数
        self.grid()                             # 调用组件的 grid()方法,调整其显示位置和
                                                # 大小
        self.createWidgets()                    # 调用对象方法,创建子组件
    def createWidgets(self):                    # 对象方法:创建子组件
        # 创建 Scale 组件
        self.scaleFont = tk.Scale(self, from_ = 10, to = 60, length = 400,
        orient = tk.HORIZONTAL, command = self.changefont)
        self.scaleFont.set(20)                  # 设置初始值
        self.scaleFont.pack()                   # 调用 pack()方法,调整其显示位置和大小
        self.lblTitle = tk.Label(self, text = 'Hello', font = ('Helvetica', 20, 'bold'))
                                                # 创建 Label 组件
        self.lblTitle.pack()                    # 调用 pack()方法,调整其显示位置和大小
    def changefont(self, value):                # 定义事件处理程序:改变字体
        fontNew = ('Helvetica', self.scaleFont.get(), 'bold')
        self.lblTitle.config(font = fontNew)
root = tk.Tk()                                  # 创建一个 Tk 根窗口组件 root
root.title('设置字体大小')                       # 设置窗口标题
root['width'] = 400; root['height'] = 50        # 设置窗口的宽和高
app = Application(master = root)                # 创建 Application 的对象实例
app.mainloop()                                  # 调用组件的 mainloop()方法,进入事件循环
```

12.5.12 Toplevel

Toplevel(顶层窗口)是直接由窗口管理器管理的窗口，顶层窗口独立于其他窗口，可以创建任意数量的顶层窗口。

```
            fontNew = ('Helvetica', self.vFont.get(), 'bold')
            self.lblTitle.config(font = fontNew)
root = tk.Tk()                                  #创建一个 Tk 根窗口组件 root
root.title('设置字体大小')                       #设置窗口标题
root['width'] = 400; root['height'] = 50        #设置窗口的宽、高
app = Application(master = root)                #创建 Application 的对象实例
app.mainloop()                                  #调用组件的 mainloop()方法,进入事件循环
```

12.5.11 Scale

Scale(移动滑块)控件用于在有界区间内通过移动滑块来选择值。

【例 12.23】 Scale 示例(Scale.py):移动滑块,改变字体大小。程序运行效果如图 12-22 所示。



图 12-22 Scale 程序运行效果

```
import tkinter as tk                            #导入 tkinter 模块
class Application(tk.Frame):                    #定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master = None):          #构造函数,master 为父窗口
        tk.Frame.__init__(self, master)         #调用父类的构造函数
        self.grid()                             #调用组件的 grid()方法,调整其显示位置和
                                                #大小
        self.createWidgets()                    #调用对象方法,创建子组件
    def createWidgets(self):                    #对象方法:创建子组件
        #创建 Scale 组件
        self.scaleFont = tk.Scale(self, from_ = 10, to = 60, length = 400,
        orient = tk.HORIZONTAL, command = self.changefont)
        self.scaleFont.set(20)                  #设置初始值
        self.scaleFont.pack()                   #调用 pack()方法,调整其显示位置和大小
        self.lblTitle = tk.Label(self, text = 'Hello', font = ('Helvetica', 20, 'bold'))
                                                #创建 Label 组件
        self.lblTitle.pack()                    #调用 pack()方法,调整其显示位置和大小
    def changefont(self, value):                #定义事件处理程序:改变字体
        fontNew = ('Helvetica', self.scaleFont.get(), 'bold')
        self.lblTitle.config(font = fontNew)
root = tk.Tk()                                  #创建一个 Tk 根窗口组件 root
root.title('设置字体大小')                       #设置窗口标题
root['width'] = 400; root['height'] = 50        #设置窗口的宽和高
app = Application(master = root)                #创建 Application 的对象实例
app.mainloop()                                  #调用组件的 mainloop()方法,进入事件循环
```

12.5.12 Toplevel

Toplevel(顶层窗口)是直接由窗口管理器管理的窗口,顶层窗口独立于其他窗口,可以创建任意数量的顶层窗口。

【例 12.24】 使用 Toplevel 实现自定义关于对话框(MyDialog.py),程序运行效果如图 12-23 所示。



图 12-23　自定义关于对话框的程序运行效果

```
import tkinter as tk                                    #导入 tkinter 模块
class MyDialog:                                         #自定义对话框
    def __init__(self, master):                         #构造函数
        self.top = tk.Toplevel(master)                  #生成 Toplevel 组件
        self.label1 = tk.Label(self.top, text='版权所有')    #创建标签组件
        self.label1.pack()                              #调用 pack()方法,调整其显示位置和大小
        self.label2 = tk.Label(self.top, text='V 1.0.0')   #创建标签组件
        self.label2.pack()                              #调用 pack()方法,调整其显示位置和大小
        self.buttonOK = tk.Button(self.top, text='OK', command=self.funcOk)
                                                        #创建按钮
        self.buttonOK.pack()                            #调用 pack()方法,调整其显示位置和大小
    def funcOk(self):
        self.top.destroy()                              #销毁对话框
class Application(tk.Frame):                            #定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master=None):                    #构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                 #调用父类的构造函数
        self.pack()                                     #调用组件的 pack()方法,调整其显示位置和
                                                        #大小
        self.createWidgets()                            #调用对象方法,创建子组件
    def createWidgets(self):                            #对象方法:创建子组件
        self.btnAbout = tk.Button(self, text="About", command=self.funcAbout)
                                                        #创建 Button 组件
        self.btnAbout.pack()                            #调用组件的 pack()方法,调整其显示位置和
                                                        #大小
    def funcAbout(self):                                #定义事件处理程序
        d = MyDialog(self)                              #创建对话框
root = tk.Tk()                                          #创建一个 Tk 根窗口组件 root
root['width']=400; root['height'] = 50                  #设置窗口的宽、高
app = Application(master=root)                          #创建 Application 的对象实例
app.mainloop()                                          #调用组件的 mainloop()方法,进入事件循环
```

12.5.13　ttk 子模块控件

tkinter 模块包括子模块 ttk,ttk 包含了 tkinter 缺少的基本控件 Combobox、Progressbar、Notebook、Treeview 等,使 tkinter 更实用。ttk 还支持控件呈现操作系统本地化风格,在 Windows 下像 Windows,在 Mac OS X 下像 Mac,在 Linux 下像 Linux。

限于篇幅,本书没有展开阐述,具体内容,请读者查看 tkinter 参考手册。

12.6　对　话　框

对话框用于与用户交互和检索信息。tkinter 模块中的子模块 messagebox、filedialog、colorchooser、simpledialog 包括一些通用的预定义对话框;用户也可以通过继承 TopLevel 创建自定义对话框。

12.6.1 通用消息对话框

tkinter 模块的子模块 messagebox 中包含如下若干用于打开消息对话框的函数。

- askokcancel(title=None，message=None， ** options)：OK/Cancel 对话框。
- askquestion(title=None，message=None， ** options)：Yes/No 问题对话框。
- askretrycancel(title=None，message=None， ** options)：Retry/Cancel 对话框。
- askyesno(title=None，message=None， ** options)：Yes/No 对话框。
- showerror(title=None，message=None， ** options)：错误消息对话框。
- showinfo(title=None，message=None， ** options)：信息消息对话框。
- showwarning(title=None，message=None， ** options)：警告消息对话框。

其中，title 是弹出对话框窗口的标题；message 是对话框中显示的内容，使用转义字符“\n”可多行显示。命名参数 options 指定如下选项。

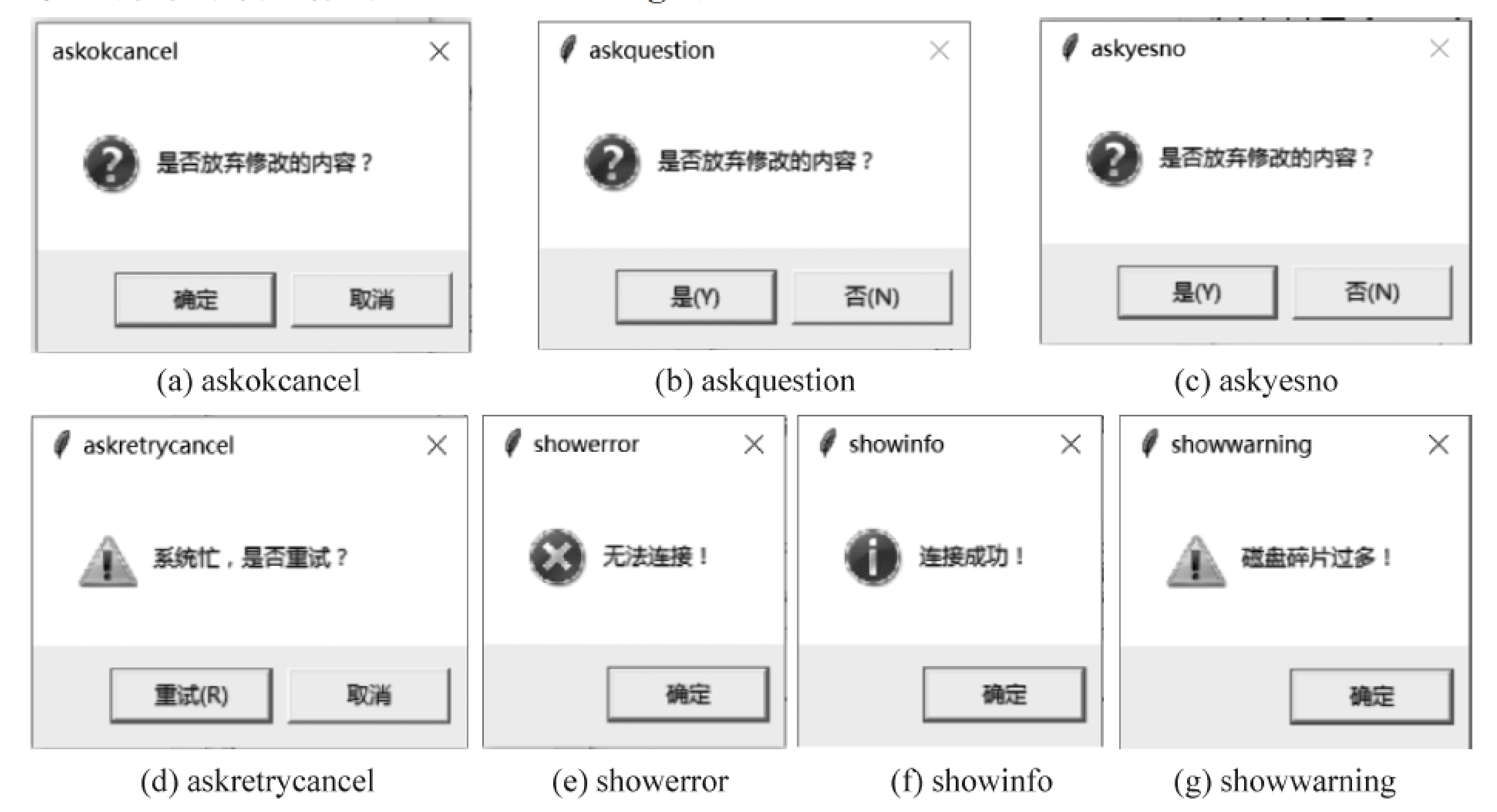
- default=C：默认按钮，取值为模块常量 CANCEL、IGNORE、OK、NO、RETRY、YES。默认为 CANCEL 按钮。
- icon=I：图标，取值为模块常量 ERROR、INFO、QUESTION、WARNING。
- parent=W：父窗口，默认为根窗口。

askokcancel、askretrycancel 和 askyesno 返回 bool 值，当单击 OK 或 Yes 按钮时返回 True，当单击 No 或 Cancel 时返回 False；askquestion 返回字符串，当单击 Yes 时返回 u'yes'，当单击 No 时返回 u'no'。

【例 12.25】 通用消息对话框示例(dialog.py)。

```
from tkinter.messagebox import *                    #导入 tkinter 模块中的子模块 messagebox
r1 = askokcancel(title = 'askokcancel', message = '是否放弃修改的内容?')
r2 = askquestion(title = 'askquestion', message = '是否放弃修改的内容?')
r3 = askyesno(title = 'askyesno', message = '是否放弃修改的内容?')
r4 = askretrycancel(title = 'askretrycancel', message = '系统忙,是否重试?')
showerror(title = 'showerror', message = '无法连接!')
showinfo(title = 'showinfo', message = '连接成功!')
showwarning(title = 'showwarning', message = '磁盘碎片过多!')
```

程序运行结果分别如图 12-24(a)～(g)所示。



(a) askokcancel　(b) askquestion　(c) askyesno

(d) askretrycancel　(e) showerror　(f) showinfo　(g) showwarning

图 12-24 通用消息对话框

12.6.2　文件对话框

tkinter 模块的子模块 filedialog 中包含如下若干用于打开文件对话框的函数。

- askdirectory(** options)：打开目录对话框，返回目录名。
- askopenfile(** options)：打开文件对话框，返回打开的文件对象。
- askopenfiles(** options)：打开文件对话框，返回打开的文件对象列表。
- askopenfilename(** options)：打开文件对话框，返回打开的文件名。
- askopenfilenames(** options)：打开文件对话框，返回打开的文件名列表。
- asksaveasfile(mode='w', ** options)：打开保存对话框，返回保存的文件对象。
- asksaveasfilename(mode='w', ** options)：打开保存对话框，返回保存的文件名。

使用命名参数 options 指定如下选项。

- defaultextension=s：默认后缀为.xxx。用户没有输入后缀时自动添加。
- filetypes=[(label1, pattern1), (label2, pattern2),…]：文件过滤器。
- initialdir=D：初始目录。
- initialfile=F：初始文件。
- parent=W：父窗口。默认为根窗口。
- title=T：窗口标题。

【例 12.26】　文件对话框示例(filedialog.py)。

```
from tkinter.filedialog import *                    #导入 tkinter 模块中的子模块 filedialog
f = askopenfilename(title = 'askopenfilename', filetypes = [('Python 源文件','.py')])
```

程序运行结果如图 12-25 所示。

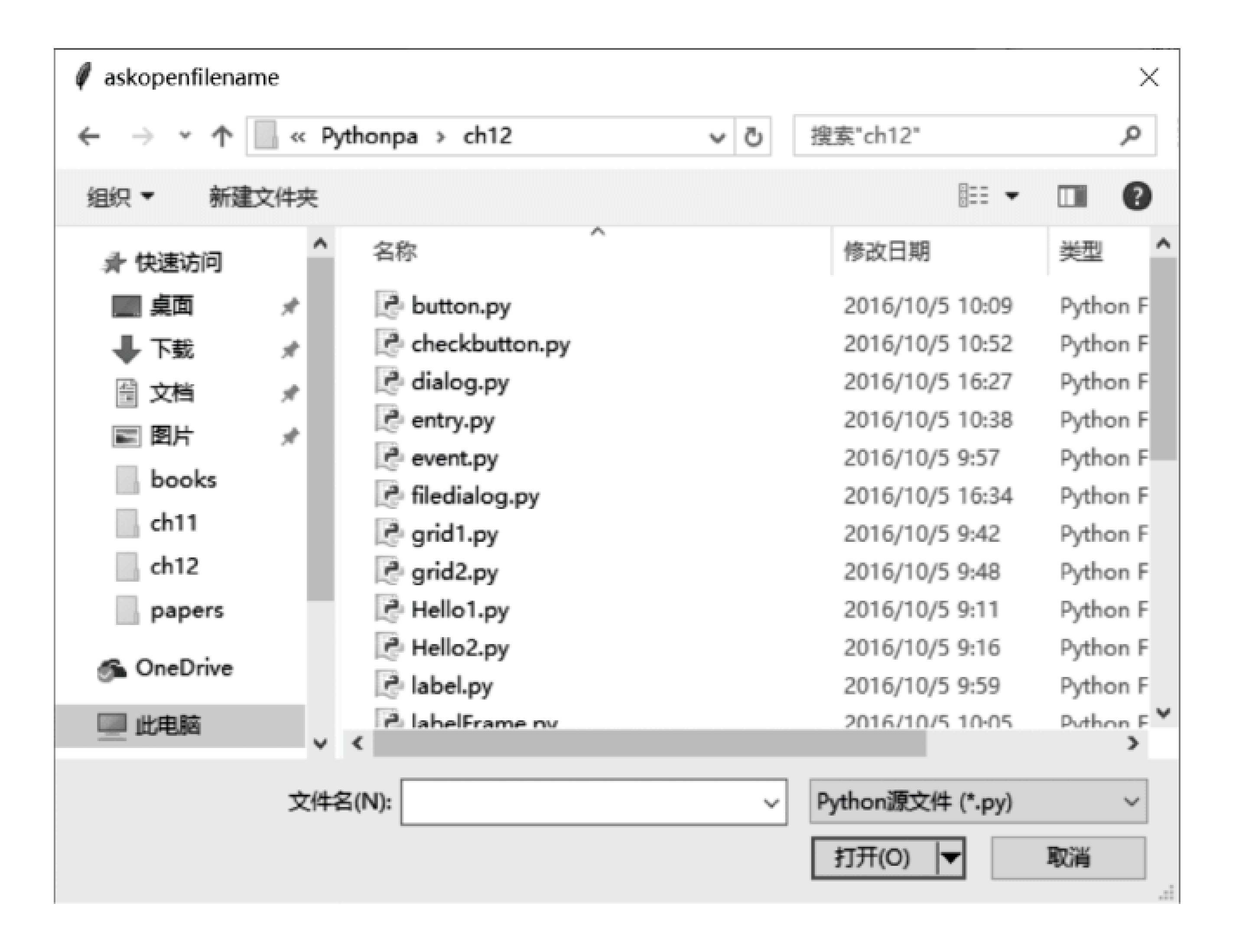


图 12-25　文件对话框示例

12.6.3　颜色选择对话框

tkinter 模块的子模块 colorchooser 中包含如下用于打开颜色选择对话框的函数：

```
askcolor(color = None, * * options)                 #打开颜色选择对话框
```

其中,color 为初始颜色;命名参数 options 指定如下选项。

- parent=W:父窗口。默认为根窗口。
- title=T:窗口标题。

askcolor()返回((R,G,B),color)。

【例 12.27】 颜色对话框示例(colordialog.py)。

```
from tkinter.colorchooser import *                    #导入 tkinter 模块中的子模块 colorchooser
c = askcolor(color='red', title='askcolor')           #((0.0, 0.0, 255.99609375), '#0000ff')
```

程序运行效果如图 12-26 所示。

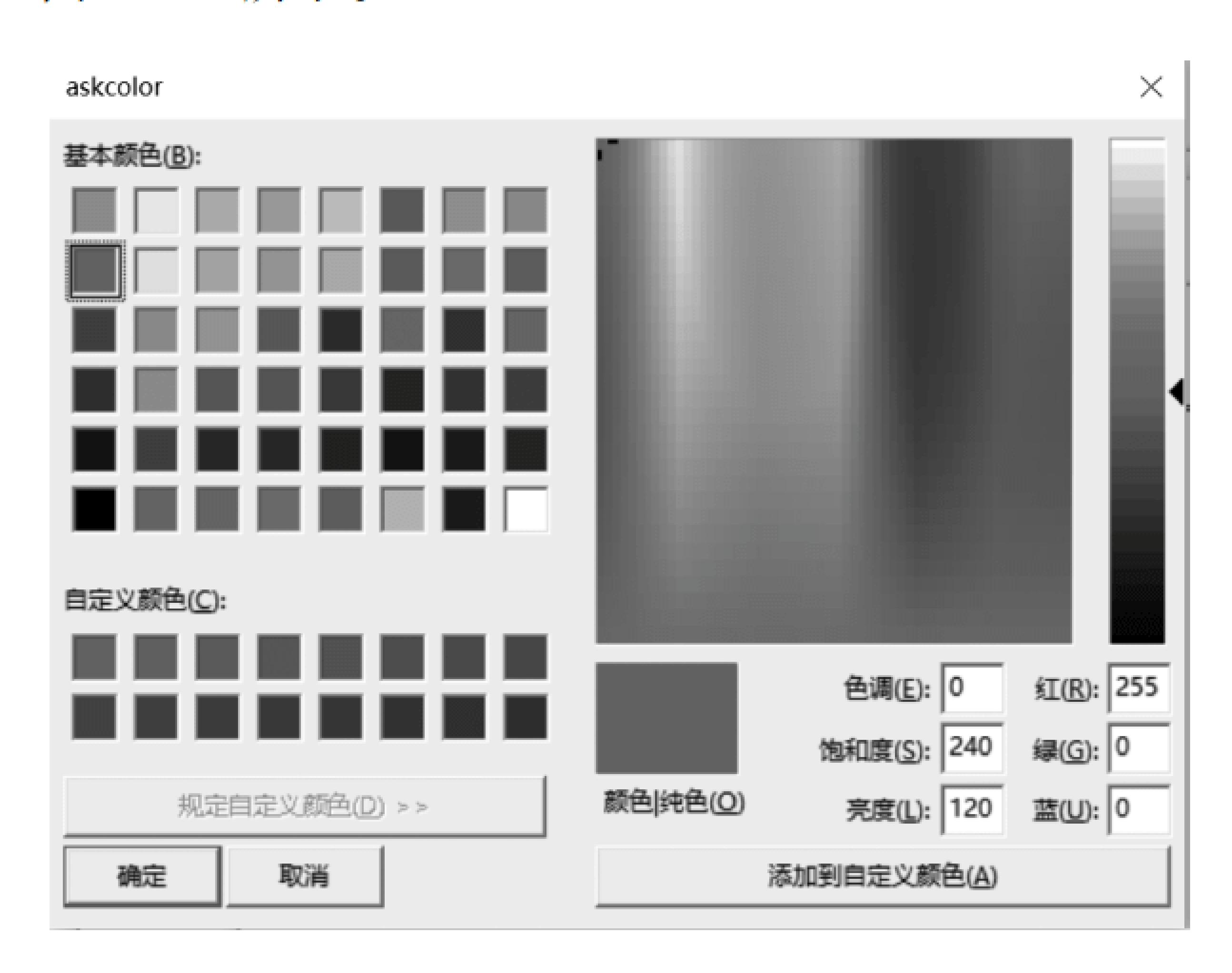


图 12-26 颜色选择对话框

JColorChooser 提供一个允许用户操作和选择颜色的控制器窗格。JColorChooser 可以作为控件放置在任何自定义的界面当中,也可以作为单独的对话框使用。

12.6.4 通用对话框应用举例

【例 12.28】 通用对话框应用示例(DialogEditor.py)。程序运行效果如图 12-27 所示。

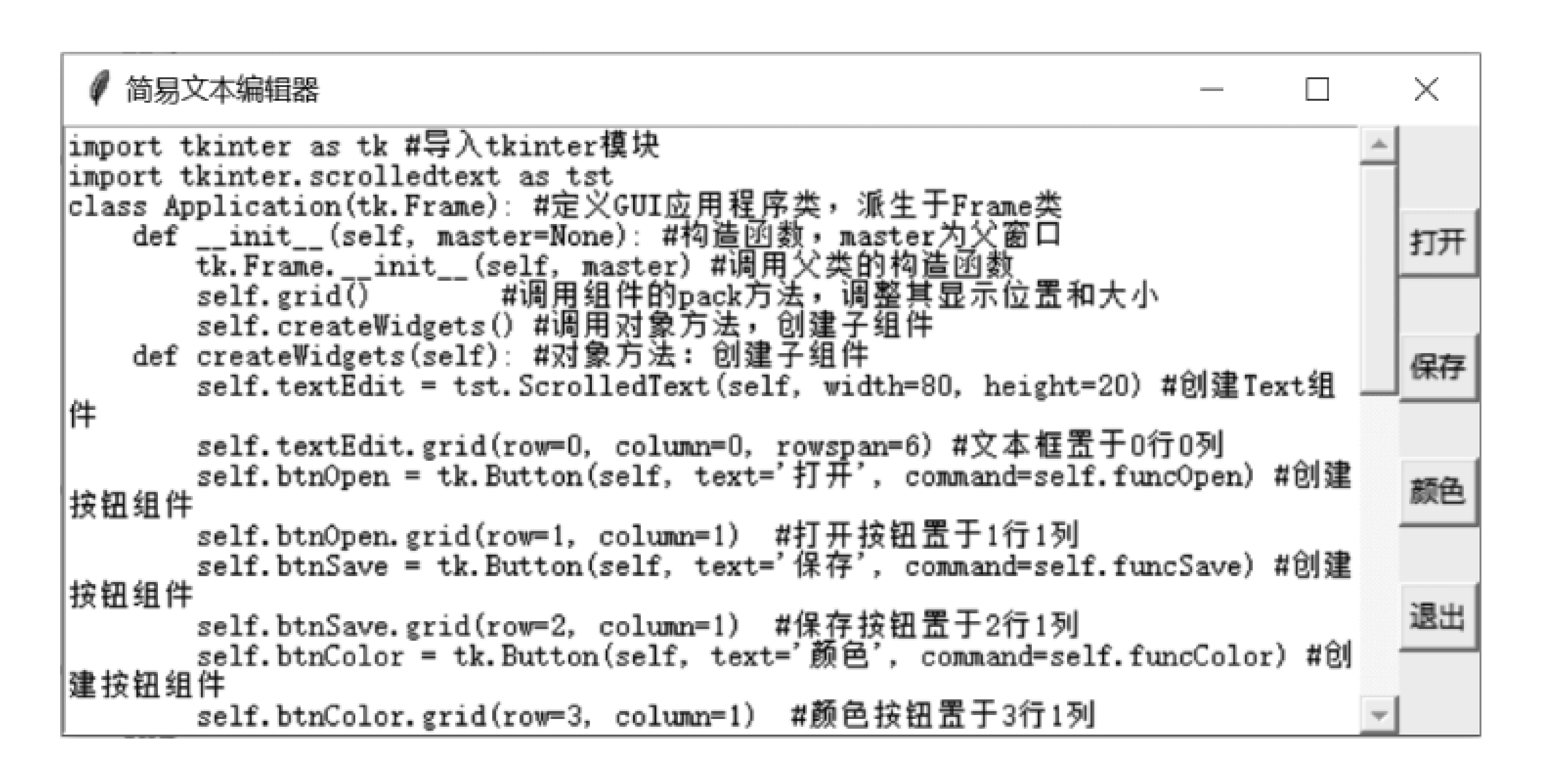


图 12-27 通用对话框的程序运行效果

```
import tkinter as tk                           #导入 tkinter 模块
from tkinter.filedialog import *               #导入 tkinter 模块中的子模块 filedialog
from tkinter.colorchooser import *             #导入 tkinter 模块中的子模块 colorchooser
import tkinter.scrolledtext as tst             #导入 tkinter 模块中的子模块 scrolledtext
```

```
class Application(tk.Frame):                      # 定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master = None):            # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)           # 调用父类的构造函数
        self.grid()                               # 调用组件的 grid()方法,调整其显示位置和大小
        self.createWidgets()                      # 调用对象方法,创建子组件
    def createWidgets(self):                      # 对象方法:创建子组件
        self.textEdit = tst.ScrolledText(self, width = 80, height = 20)
                                                  # 创建 ScrolledText 组件
        self.textEdit.grid(row = 0, column = 0, rowspan = 6)
                                                  # 文本框置于 0 行 0 列
        self.btnOpen = tk.Button(self, text = '打开', command = self.funcOpen)
                                                  # 创建打开按钮组件
        self.btnOpen.grid(row = 1, column = 1)    # 打开按钮置于 1 行 1 列
        self.btnSave = tk.Button(self, text = '保存', command = self.funcSave)
                                                  # 创建保存按钮组件
        self.btnSave.grid(row = 2, column = 1)    # 保存按钮置于 2 行 1 列
        self.btnColor = tk.Button(self, text = '颜色', command = self.funcColor)
                                                  # 创建颜色按钮组件
        self.btnColor.grid(row = 3, column = 1)   # 颜色按钮置于 3 行 1 列
        self.btnQuit = tk.Button(self, text = '退出', command = self.funcQuit)
                                                  # 创建退出按钮组件
        self.btnQuit.grid(row = 4, column = 1)    # 退出按钮置于 4 行 1 列
    def funcOpen(self):                           # 定义事件处理程序:打开文件
        self.textEdit.delete(1.0, tk.END)         # 清空 Text 组件的内容
        fname = tk.filedialog.askopenfilename(filetypes = [('Python 源文件', '.py')])
        with open(fname, 'r', encoding = 'utf-8') as f1:   # 打开文件
            str1 = f1.read()                               # 读入文件内容
        self.textEdit.insert(0.0, str1)                    # 插入内容到 Text 组件
    def funcSave(self):                                    # 定义事件处理程序:保存文件
        str1 = self.textEdit.get(1.0, tk.END)
        fname = tk.filedialog.asksaveasfilename(filetypes = [('Python 源文件', '.py')])
        with open(fname, 'w', encoding = 'utf-8') as f1:   # 打开文件
            f1.write(str1)
    def funcColor(self):                                   # 定义事件处理程序:设置颜色
        t, c = tk.colorchooser.askcolor(title = 'askcolor')
        self.textEdit.config(bg = c)
    def funcQuit(self):                                    # 定义事件处理程序:退出程序
        root.destroy()                                     # 退出程序
root = tk.Tk()                                             # 创建一个 Tk 根窗口组件 root
root.title('简易文本编辑器')                                # 设置窗口标题
app = Application(master = root)                           # 创建 Application 的对象实例
app.mainloop()                                             # 调用组件的 mainloop()方法,进入
                                                           # 事件循环
```

12.6.5　简单对话框

tkinter 模块的子模块 simpledialog 中包含如下若干用于打开输入对话框的函数。

- askfloat(title, prompt, ** kw):打开输入对话框,输入并返回浮点数。
- askinteger(title, prompt, ** kw):打开输入对话框,输入并返回整数。
- askstring(title, prompt, ** kw):打开输入对话框,输入并返回字符串。

其中,title 为窗口标题,prompt 为提示文本信息;命名参数 kw 指定各种选项,包括 initialvalue(初始值)、minvalue(最小值)和 maxvalue(最大值)。

【例 12.29】 简单对话框示例 1(sdialog1.py)。

```
from tkinter import *                              #导入 tkinter 模块的所有内容
root = Tk()                                        #创建一个 Tk 根窗口组件
from tkinter.simpledialog import *                 #导入 tkinter 模块中的子模块 simpledialog
i = askinteger(title='请输入', prompt='请输入整数:',initialvalue=100)
f = askfloat(title='请输入', prompt='请输入实数:')
s = askstring(title='请输入', prompt='请输入字符串:')
```

程序运行效果如图 12-28(a)~(c)所示。



(a) 输入整数　(b) 输入实数　(c) 输入字符串

图 12-28 简单对话框程序 1 运行效果

tkinter 模块的子模块 simpledialog 中包含 SimpleDialog 组件,其构造函数如下:

```
SimpleDialog(master, text='', buttons=[], default=None, cancel=None, title=None, class_=None)
```

其中,master 是父窗口;buttons 为要显示的按钮列表;default 为默认选中的按钮;title 为窗口标题。

【例 12.30】 简单对话框示例 2(sdialog2.py)。

```
from tkinter import *                              #导入 tkinter 模块的所有内容
root = Tk()                                        #创建一个 Tk 根窗口组件
from tkinter.simpledialog import *                 #导入 tkinter 模块中的子模块 simpledialog
#创建 SimpleDialog 组件
dlg = SimpleDialog(root, text='继续?', buttons=['Yes','No','cancel'], default = 0)
```

程序运行效果如图 12-29 所示。



图 12-29 简单对话框程序 2 运行效果

12.7 菜单和工具栏

图形用户界面应用程序通常提供菜单,菜单包括各种按照主题分组的基本命令。图形用户界面应用程序包括如下 3 种类型的菜单。

(1) 主菜单:提供窗体的菜单系统。通过单击可以下拉出子菜单,选择命令可以执行相关的操作。常用的主菜单通常包括文件、编辑、视图、帮助等。

(2) 上下文菜单(也称为快捷菜单):通过鼠标右击某对象而弹出的菜单,一般包含与该对象相关的常用菜单命令,例如剪切、复制、粘贴等。

(3) 工具栏:提供窗体的工具栏。通过单击工具栏上的图标可以执行相关的操作。

12.7.1　创建主菜单

主菜单一般提供窗体的菜单系统。通过单击可以下拉出子菜单,选择命令可以执行相关的操作。常用的主菜单通常包括文件、编辑、视图和帮助等。创建主菜单一般遵循下列步骤。

(1) 创建主菜单栏。例如:

```
menubar = tk.Menu(root)                  #创建主菜单栏 menubar
```

(2) 创建菜单,并添加菜单到步骤(1)创建的菜单栏。例如:

```
menufile = tk.Menu(menubar)              #创建菜单 menufile
#把菜单 menufile 作为层叠菜单添加到主菜单栏 menubar
menubar.add_cascade(label = 'File',menu = menufile)
```

(3) 添加菜单项到步骤(2)创建的菜单。例如:

```
menufile.add_command(label = 'Open')     #在菜单 menufile 中添加菜单项 Open
menufile.add_command(label = 'Save')     #在菜单 menufile 中添加菜单项 Save
menufile.add_command(label = 'Print', accelerator = '^P', command = f_print)
                                         #添加菜单项 Print
menufile.add_separator()                 #添加分隔符
menufile.add_command(label = 'Exit')     #添加菜单项 Exit
```

(4) 将菜单栏添加到根窗体。例如:

```
root['menu'] = menubar                   #添加主菜单到根窗口
```

【例 12.31】 主菜单示例(menu.py)。程序运行效果如图 12-30 所示。

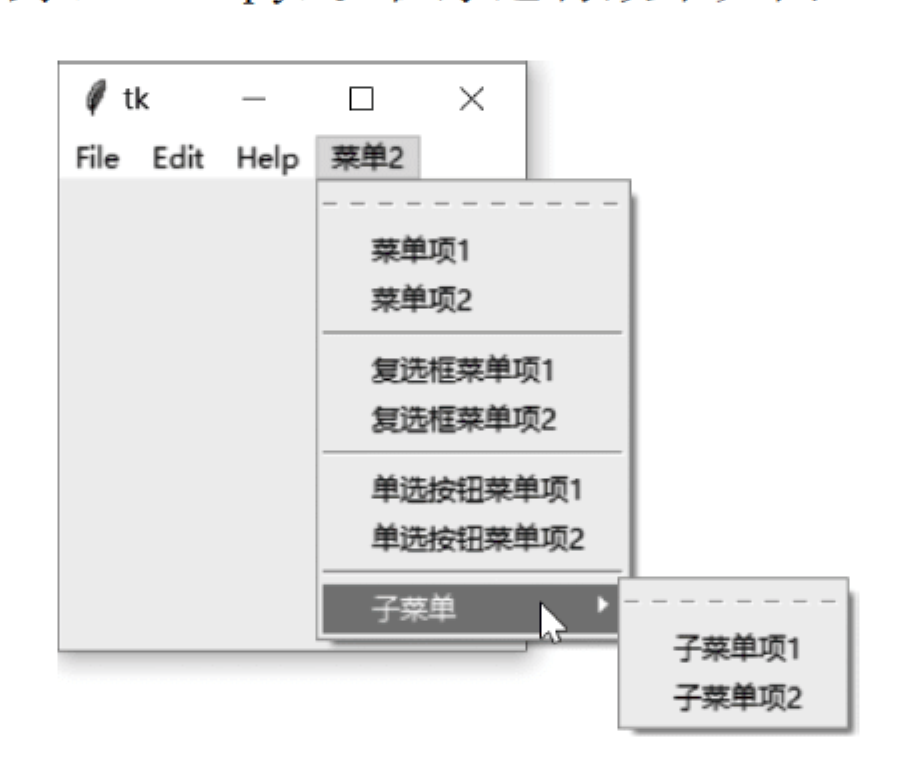


图 12-30　菜单的程序运行效果

```
import tkinter as tk                     #导入 tkinter 模块
def f_print():
    tk.messagebox.showinfo('信息', '打印功能')
root = tk.Tk()                           #创建一个 Tk 根窗口组件 root
#创建主菜单栏
menubar = tk.Menu(root)                  #创建主菜单栏 menubar
#创建子菜单
menufile = tk.Menu(menubar)              #创建菜单 menufile
menuedit = tk.Menu(menubar, tearoff = 0)#创建菜单 menuedit
menuhelp = tk.Menu(menubar, tearoff = 0)#创建菜单 menuhelp
menuTest = tk.Menu(menubar)              #创建菜单 menuTest
menubar.add_cascade(label = 'File', menu = menufile)
                                         #menufile 作为层叠菜单添加到主菜单栏
menubar.add_cascade(label = "Edit", menu = menuedit)
                                         # menuedit 作为层叠菜单添加到主菜单栏
menubar.add_cascade(label = "Help", menu = menuhelp)
                                         # menuhelp 作为层叠菜单添加到主菜单栏
```

```
menubar.add_cascade(label = "菜单 2", menu = menuTest)
                                        # menuTest 作为层叠菜单添加到主菜单栏
#添加菜单项
menufile.add_command(label = 'Open')    #在菜单 menufile 中添加菜单项 Open
menufile.add_command(label = 'Save')    #添加菜单项 Save
menufile.add_command(label = 'Print', accelerator = '^P', command = f_print)
                                        #添加菜单项 Print
menufile.add_separator()                #添加分隔符
menufile.add_command(label = 'Exit')    #添加菜单项 Exit
menuedit.add_command(label = "Cut")     #在菜单 menuedit 中添加菜单项 Cut
menuedit.add_command(label = "Copy")    #添加菜单项 Copy
menuedit.add_command(label = "Paste")   #添加菜单项 Paste
menuhelp.add_command(label = "About")   #在菜单 menuhelp 中添加菜单项 About
menuTest.add_command(label = "菜单项 1")#在菜单 menuTest 中添加菜单项 1
menuTest.add_command(label = "菜单项 2")#添加菜单项 2
menuTest.add_separator()                #添加分隔符
menuTest.add_checkbutton(label = "复选框菜单项 1")
                                        #添加复选框菜单项 1
menuTest.add_checkbutton(label = "复选框菜单项 2")
                                        #添加复选框菜单项 2
menuTest.add_separator()                #添加分隔符
menuTest.add_radiobutton(label = "单选按钮菜单项 1")
                                        #添加单选按钮菜单项 1
menuTest.add_radiobutton(label = "单选按钮菜单项 2")
                                        #添加单选按钮菜单项 2
menuTest.add_separator()                #添加分隔符
menusub  =  tk.Menu(menuTest)           #创建子菜单
menuTest.add_cascade(label = "子菜单", menu = menusub)
                                        #menusub 作为层叠菜单添加到菜单
menusub.add_command(label = "子菜单项 1") #添加子菜单项 1
menusub.add_command(label = "子菜单项 2") #添加子菜单项 2
#附加主菜单到根窗口
root['menu']  =  menubar                #附加主菜单到根窗口
root.mainloop()                         #调用组件的 mainloop()方法,进入事件循环
```

12.7.2 创建上下文菜单

上下文菜单(也称为快捷菜单)是通过鼠标右击某对象而弹出的菜单,一般包含与该对象相关的常用菜单命令,例如剪切、复制、粘贴等。创建上下文菜单一般遵循下列步骤。

(1) 创建菜单(与创建主菜单相同)。例如:

```
menubar  =  tk.Menu(root)
menubar.add_command(label = "Font")
```

(2) 绑定鼠标右击事件,并在事件处理函数中弹出菜单。例如:

```
def popup(event):                                  #事件处理函数
    menubar.post(event.x_root, event.y_root)       #在鼠标右键位置显示菜单
root.bind('<Button-3>', popup)                     #绑定事件
```

【例 12.32】 上下文菜单示例(popmenu.py)。程序运行效果如图 12-31 所示。

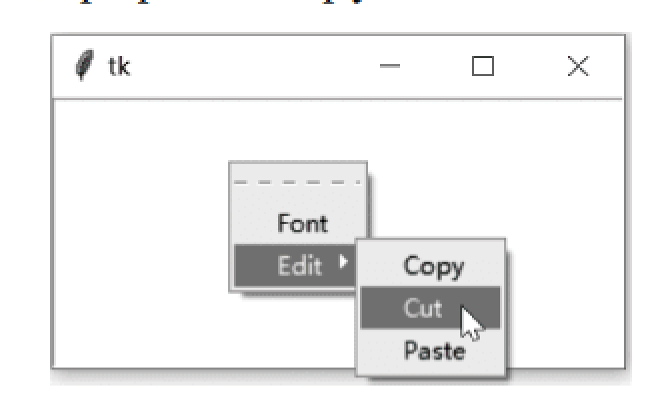


图 12-31 上下文菜单程序运行效果

```
import tkinter as tk                                    # 导入 tkinter 模块
def popup(event):                                       # 事件处理函数
    menubar.post(event.x_root, event.y_root)            # 在鼠标右键位置显示菜单
root = tk.Tk()                                          # 创建一个 Tk 根窗口组件 root
# 创建菜单
menubar = tk.Menu(root)                                 # 创建菜单
menubar.add_command(label = "Font")                     # 添加 Font 命令
menuedit = tk.Menu(menubar, tearoff = 0)                # 创建菜单 menuedit
menubar.add_cascade(label = "Edit", menu = menuedit)
                                                        # menuedit 作为层叠菜单添加到上下文菜单
menuedit.add_command(label = "Copy")                    # 添加 Copy 命令
menuedit.add_command(label = "Cut")                     # 添加 Cut 命令
menuedit.add_command(label = "Paste")                   # 添加 Paste 命令
# 创建界面
textEdit = tk.Text(root, width = 40, height = 10)# 创建 Text 组件
textEdit.pack()                                         # 调用 pack()方法,调整其显示位置和大小
root.bind('<Button - 3>', popup)                        # 绑定事件
# 添加主菜单到根窗口
root.mainloop()                                         # 调用组件的 mainloop()方法,进入事件循环
```

12.7.3 菜单应用举例

【例 12.33】 简单文本编辑器示例(MenuEditor.py)。程序运行效果如图 12-32 所示。

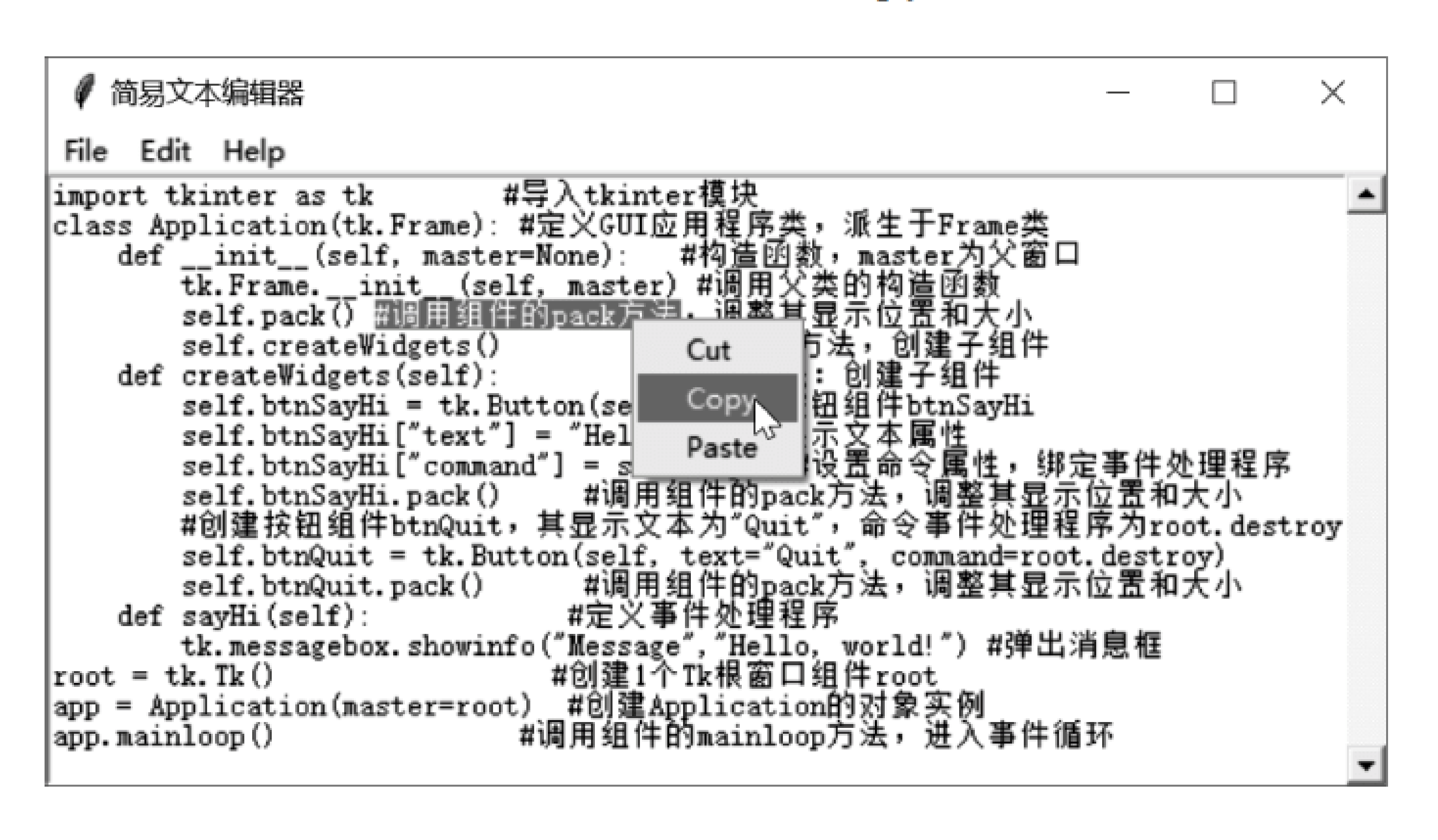


图 12-32 简单文本编辑器的程序运行效果

```
import tkinter as tk                                    # 导入 tkinter 模块
from tkinter.filedialog import *                        # 导入 tkinter 模块中的子模块 filedialog
from tkinter import messagebox                          # 导入 tkinter 模块中的子模块 messagebox
import tkinter.scrolledtext as tst                      # 导入 tkinter 模块中的子模块 scrolledtext
class Application(tk.Frame):                            # 定义 GUI 应用程序类,派生于 Frame 类
    def __init__(self, master = None):                  # 构造函数,master 为父窗口
        tk.Frame.__init__(self, master)                 # 调用父类的构造函数
        self.grid()                                     # 调用组件的 grid()方法,调整其显示位置和大小
        self.createWidgets()                            # 调用对象方法,创建子组件
        self.createMenu()                               # 调用对象方法,创建菜单
        root['menu'] = self.menubar                     # 添加主菜单到根窗口
        root.bind('<Button - 3>', self.f_popup) # 绑定事件
    def createWidgets(self):                            # 对象方法:创建子组件
        self.textEdit = tst.ScrolledText(self, width = 80, height = 20)
                                                        # 创建 ScrolledText 组件
        self.textEdit.grid(row = 0, column = 0, rowspan = 6)
```

```
                                            #Text 组件置于 0 行 0 列,跨 6 行
def createMenu(self):                       #对象方法:创建菜单
    self.menubar = tk.Menu(root)            #创建主菜单栏 menubar
    #创建子菜单
    self.menufile = tk.Menu(self.menubar)                           #创建菜单 menufile
    self.menuedit = tk.Menu(self.menubar, tearoff=0)                #创建菜单 menuedit
    self.menuhelp = tk.Menu(self.menubar, tearoff=0)                #创建菜单 menuhelp
    self.menubar.add_cascade(label='File', menu=self.menufile)      #添加菜单项'File'
    self.menubar.add_cascade(label="Edit", menu=self.menuedit)      #添加菜单项'Edit'
    self.menubar.add_cascade(label="Help", menu=self.menuhelp)      #添加菜单项'Help'
    #添加菜单项
    self.menufile.add_command(label='New', command=self.f_new)      #File-New
    self.menufile.add_command(label='Open', command=self.f_open)    #File-Open
    self.menufile.add_command(label='Save', accelerator='^A',
                              command=self.f_save)                  #File-Save
    self.menufile.add_separator()                                   #分隔符
    self.menufile.add_command(label='Exit', command=root.destroy)   #File-Exit
    self.menuedit.add_command(label="Cut", command=self.f_cut)      #Edit-Cut
    self.menuedit.add_command(label="Copy", command=self.f_copy)    #Edit-Copy
    self.menuedit.add_command(label="Paste", command=self.f_paste)  #Edit-Paste
    self.menuhelp.add_command(label="About", command=self.f_about)  #Help-About
def f_new(self):                            #定义事件处理程序:File-New
    self.textEdit.delete(1.0, tk.END)       #清空 Text 组件的内容
def f_open(self):                           #定义事件处理程序:File-Open
    self.textEdit.delete(1.0, tk.END)       #清空 Text 组件的内容
    fname = tk.filedialog.askopenfilename(filetypes=[('Python 源文件','.py')])
    with open(fname, 'r', encoding= 'utf-8') as f1:   #打开文件
        str1 = f1.read()                    #读入文件内容
    self.textEdit.insert(0.0, str1)         #插入内容到 Text 组件
def f_save(self):                           #定义事件处理程序:File-Save
    str1 = self.textEdit.get(1.0, tk.END)   #获取 Text 组件中的全部内容
    fname = tk.filedialog.asksaveasfilename(filetypes=[('Python 源文件','.py')])
    with open(fname, 'w', encoding= 'utf-8') as f1:  #打开文件
        f1.write(str1)                      #将 Text 组件中的全部内容写入文件
def f_about(self):                          #定义事件处理程序:Help-About
    tk.messagebox.showinfo('关于', '版本 V 1.0.1')
def f_cut(self):                            #定义事件处理程序:Edit-Cut
    try:
        str1=self.textEdit.get(tk.SEL_FIRST, tk.SEL_LAST)
                                            #获取选择的内容
        self.textEdit.clipboard_clear()     #清空剪贴板
        self.textEdit.clipboard_append(str1) #添加到剪贴板
        self.textEdit.delete(tk.SEL_FIRST, tk.SEL_LAST)
                                            #删除选择的内容
    except: pass
def f_copy(self):                           #定义事件处理程序:Edit-Copy
    try:
        str1=self.textEdit.get(tk.SEL_FIRST, tk.SEL_LAST)
                                            #获取选择的内容
        self.textEdit.clipboard_clear()     #清空剪贴板
        self.textEdit.clipboard_append(str1) #添加到剪贴板
    except: pass
def f_paste(self):                          #定义事件处理程序:Edit-Paste
    str1 = self.textEdit.selection_get(selection='CLIPBOARD')
                                            #获取剪贴板内容
```

```
        try:                                          #使用剪贴板内容替换所选内容,否则插入
                                                      #剪贴板内容
            self.textEdit.replace(tk.SEL_FIRST, tk.SEL_LAST, str1)
        except:
            self.textEdit.insert(tk.INSERT, str1)     #插入内容到当前位置
    def f_popup(self, event):                         #事件处理函数
        self.menuedit.post(event.x_root, event.y_root)
                                                      #在鼠标右键位置显示菜单
root = tk.Tk()                                        #创建一个 Tk 根窗口组件 root
root.title('简易文本编辑器')                          #设置窗口标题
app = Application(master=root)                        #创建 Application 的对象实例
app.mainloop()                                        #调用组件的 mainloop()方法,进入事件循环
```

12.8　基于 wxPython 的图形用户界面设计入门

本节使用 wxPython 实现一个简易文本编辑器,介绍使用 wxPython 开发图形界面应用程序的基本方法。

12.8.1　wxPython 概述

wxPython 是优秀的跨平台 GUI 库 wxWidgets 的 Python 封装,使用 wxPython 可以很方便地创建完整的、功能健全的图形用户界面应用程序。

wxWidgets 是一个相当稳定、高效、面向对象的 GUI 库,可以运行在 Windows、UNIX(GTK/Motif/Lesstif)和 Macintosh 平台上。因此,wxPython 是一个可以开发跨平台的图形用户界面应用程序的编程框架,对应于其他 GUI 编程框架,例如 pyQT、pyGTK 和 Tkinter 等。

12.8.2　安装 wxPython 库

【例 12.34】　使用 pip 安装 Pillow 库。

以管理员身份运行命令行提示符,通过下列命令行命令可以从 PyPI 直接安装或者更新 wxPython 库:

```
C:\WINDOWS\system32>pip3 install -U wxPython
```

使用下列 Python 语句可以查看所安装的 wxPython 版本:

```
>>> import wx
>>> print(wx.version())                    #输出:4.0.3 msw (phoenix) wxWidgets 3.0.5
```

12.8.3　wxPython 应用程序的基本架构

wxPython 应用程序的基本架构如下:

```
import wx
class MyFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, title=title, size=(400, 400))
        #创建各种组件
class App(wx.App):
    def OnInit(self):
        #创建框架窗口对象
```

```
            frame = wx.Frame(parent = None, title = 'wxPythonApplication')
            frame.Show()                                    #显示窗口
            frame.Center()                                  #窗口居中
            return True
app = App()                                                 #创建应用程序对象
app.MainLoop()                                              #进入事件循环
```

12.8.4 使用 wxPython 开发简易文本编辑器

程序实现的基本思维和关键技术方法如下。

（1）导入 wx 包。

（2）定义顶层窗口框架类（继承 wx.Frame），可以在构造函数中创建各种组件（界面元素，实现用户界面），绑定事件程序；定义方法实现事件处理函数（实现功能处理）。

（3）定义应用程序类（继承 wx.App）。在其 OnInit(self)方法中创建框架类的实例对象并显示。

（4）创建应用程序类的实例对象：app = App()。

（5）调用 app.MainLoop()，进入事件循环。

【例 12.35】 基于 wxPython 的简易文本编辑器示例程序（wxEditor.py）。程序运行效果如图 12-33 所示。

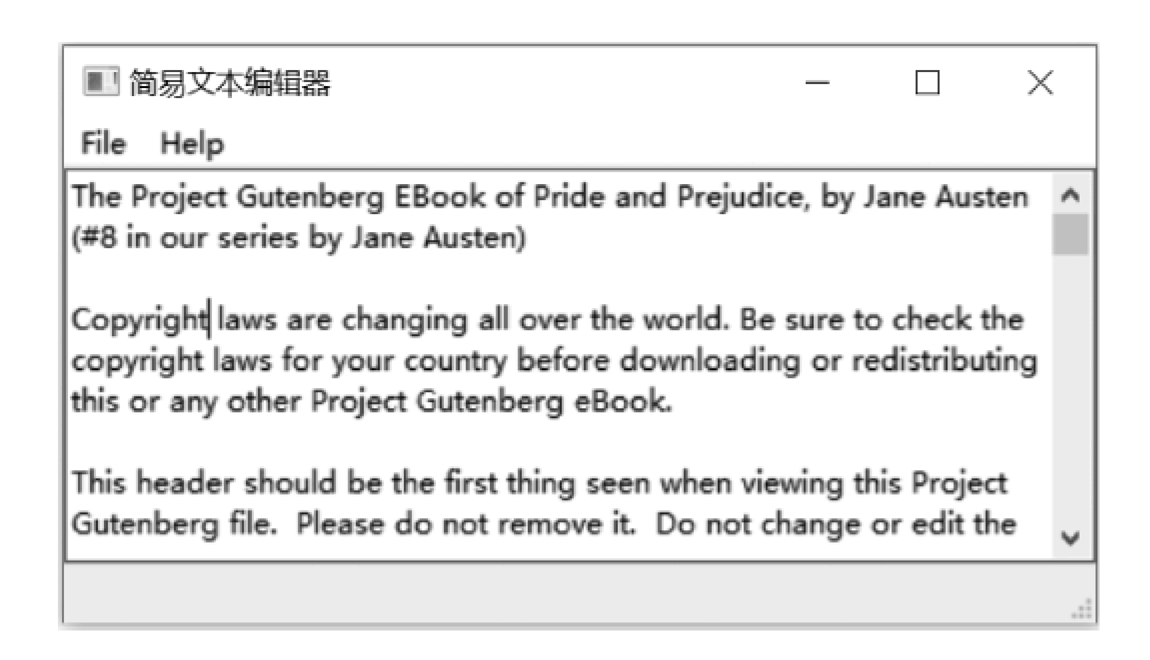


图 12-33 基于 wxPython 的简易文本编辑器

```
import os
import wx
class MainWindow(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, title = title, size = (640, 480))
        self.control = wx.TextCtrl(self, style = wx.TE_MULTILINE)
                                                    #创建多行文本控件
        self.CreateStatusBar()                      #创建状态栏
        #创建菜单并添加菜单项
        filemenu = wx.Menu()
        helpmenu = wx.Menu()
        #wx.ID_OPEN、wx.ID_SAVE、wx.ID_ABOUT、wx.ID_EXIT 是标准菜单 ID
        menuOpen = filemenu.Append(wx.ID_OPEN, "&Open", "打开")
        menuSave = filemenu.Append(wx.ID_SAVE, "&Save", "保存")
        menuExit = filemenu.Append(wx.ID_EXIT, "E&xit", "退出")
        menuAbout = helpmenu.Append(wx.ID_ABOUT, "&About", "关于")
        #创建菜单栏
        menuBar = wx.MenuBar()
        menuBar.Append(filemenu, "&File")           #把菜单"filemenu"添加到菜单栏
```

```
        menuBar.Append(helpmenu, "&Help")          #把菜单"helpmenu"添加到菜单栏
        self.SetMenuBar(menuBar)                   #把菜单栏添加到顶层框架窗口
        #绑定事件
        self.Bind(wx.EVT_MENU, self.OnOpen, menuOpen)
        self.Bind(wx.EVT_MENU, self.OnSave, menuSave)
        self.Bind(wx.EVT_MENU, self.OnExit, menuExit)
        self.Bind(wx.EVT_MENU, self.OnAbout, menuAbout)
    def OnAbout(self,e):
        """事件处理函数:显示消息对话框"""
        dlg = wx.MessageDialog(self, "简易文本编辑器 V1.0.0\nby Hong Jiang", "简易文本编辑
                        器", wx.OK)
        dlg.ShowModal()                            #显示模式对话框
        dlg.Destroy()                              #销毁对话框
    def OnExit(self,e):
        self.Close(True)                           #关闭顶层框架窗口
    def OnOpen(self, e):
        """ 事件处理函数:打开文件 """
        self.dirname = ''
        dlg = wx.FileDialog(self, "选择文件", self.dirname, "", "*.*", wx.FD_OPEN)
        if dlg.ShowModal() == wx.ID_OK:
            self.filename = dlg.GetFilename()
            self.dirname = dlg.GetDirectory()
            f = open(os.path.join(self.dirname, self.filename), 'r')
            self.control.SetValue(f.read())
            f.close()
        dlg.Destroy()
    def OnSave(self, e):
        """ 事件处理函数:保存文件 """
        self.dirname = ''
        dlg = wx.FileDialog(self, "选择文件", self.dirname, "", "*.*", wx.FD_SAVE)
        if dlg.ShowModal() == wx.ID_OK:
            self.filename = dlg.GetFilename()
            self.dirname = dlg.GetDirectory()
            f = open(os.path.join(self.dirname, self.filename), 'w')
            f.write(self.control.GetValue())
            f.close()
        dlg.Destroy()
class App(wx.App):
    def OnInit(self):
        frame = MainWindow(parent=None, title='简易文本编辑器')
        frame.Show()
        frame.Center()                             #窗口居中
        return True
app = App()
app.MainLoop()                                     #调用组件的 mainloop()方法,进入事件循环
```

12.9 复 习 题

一、填空题

1. Python 的标准 GUI 库 tkinter 由若干的模块组成,例如________、________和________等。

2. Python 图形用户界面程序一般包含一个顶层窗口,也称________或________。

3. tkinter 提供了 3 种不同的几何布局管理类，即________、________和________，用于组织和管理父组件中子配件的布局方式。

4. 通过组件的________和________选项可以设置组件的宽度和高度。

5. 通过组件的________选项可以设置其显示的文本的字体。

6. 通过组件的________选项可以设置内容停靠位置。

7. 通过组件的________选项可以设置鼠标经过组件时的光标形状。

8. 通过组件的________选项可以设置其显示的内容。通过________选项可以指定多少单位后开始换行，即显示多行；通过________选项可以指定多行的对齐方式。

9. 通过组件的________选项可以设置其显示的位图。自定义位图为________格式的文件。

10. 通过组件的________选项可以设置其显示的图像。

11. 通过组件的________选项可以设置其同时显示文本和位图/图像。

12. 通过组件的________选项可以设置其 3D 显示样式。通过________选项可以设置其鼠标经过时的 3D 显示样式。

13. 通过组件的________或________选项可以设置其边框宽度。

14. 通过组件的________和________选项可以设置其显示内容与边框之间的填充宽度和高度。

15. 通过组件的________选项可以设置其启用或禁用状态。

16. 通过组件的________选项可以设置组件显示文本时第几个字符加下画线。

17. 通过组件的________选项可以绑定 StringVar 对象到组件。

18. ________控件用于选择同一组单选按钮中的一个单选按钮（不能同时选择多个），可显示文本，也可显示图像。

19. ________控件用于选择一个或多个选项（可以同时选择多个），可显示文本，也可显示图像。

20. ________用于显示对象列表，并且允许用户选择一项或多项。

21. ________允许用户选择一个项的列表框（在用户请求时显示）。用户单击下拉按钮可显示列表框，选择的内容会显示在顶部文本框中。

22. ________控件用于在有界区间内通过移动滑块来选择值。

23. tkinter 模块中的子模块________、________、________和________包括通用的预定义对话框；用户也可以通过继承 TopLevel 创建自定义对话框。

24. tkinter 模块中的子模块________用于实现通用消息对话框的功能。

25. tkinter 模块中的子模块________用于实现文件对话框的功能。

26. tkinter 模块中的子模块________用于实现颜色选择对话框的功能。

27. tkinter 模块中的子模块________用于实现输入对话框的功能。

二、思考题

1. 在 Python 中有哪几种导入 tkinter 模块的方法？

2. 在 Python 中包括哪些常用的组件？

3. Python 图形用户界面应用程序包括哪几种类型的菜单？

4. 在 Python 中创建主菜单一般遵循哪些步骤？

5. 在 Python 中创建上下文菜单一般遵循哪些步骤？

12.10　上机实践

完成本章中的例 12.1～例 12.35，熟悉 Python 基于图形化用户界面的程序设计。

12.11　案例研究：简易图形用户界面计算器

本章案例研究为一个基于 tkinter 的简易图形用户界面计算器的设计和实现，帮助读者深入了解图形界面应用程序的开发流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 13 章

图形绘制


视频讲解

Python 提供了丰富的图形绘制功能。本章主要讲述基于 tkinter 模块的绘图、基于 turtle 模块的绘图和基于 Matplotlib 模块的绘图。

13.1 Python 绘图模块概述

在 Python 标准库中包括 tkinter(画布绘图)和 turtle(海龟绘图)两类图形绘制相关模块。

常用的开源绘图模块库如下。

(1) Matplotlib(官网"http://matplotlib.sourceforge.net/"):Matplotlib 是一个由 John Hunter 等开发的、用于绘制二维图形的 Python 模块。它利用了 Python 下的数值计算模块 Numeric 及 Numarray,复制了许多 MATLAB 中的函数,用于帮助用户轻松地获得高质量的二维图形。使用 Matplotlib 可以绘制多种形式的图形,包括普通的线图、直方图、饼图、散点图以及误差线图等;还可以比较方便地定制图形的各种属性,例如图形的类型、颜色、粗细以及字体的大小等。Matplotlib 能够很好地支持一部分 TeX 排版命令,可以比较美观地显示图形中的数学公式。

(2) Chaco:Chaco 是一个 2D 的绘图库,官网地址为"http://code.enthought.com/chaco/"。

(3) Python Google Chart:Python Google Chart 是 Google Chart API 的一个完整封装,官网地址为"http://pygooglechart.slowchop.com/"。

(4) PyCha:PyCha 是 Cairo 类库的一个简单封装和优化,官网地址为"https://bitbucket.org/lgs/pycha/wiki/Home"。

(5) pyOFC2:pyOFC2 是 Open Falsh Library 的 Python 类库,官网地址为"http://btbytes.github.com/pyofc2/"。

(6) Pychart:Pychart 用于创建高品质封装的 PostScript、PDF、PNG 或 SVG 图表 Python 库,官网地址为"http://home.gna.org/pychart/"。

(7) PLPlot:PLPlot 是用于创建科学图表的跨平台软件包,以 C 类库为核心,支持各种语言(C、C++、Fortran、Java、Python 等)。其官网地址为"http://plplot.sourceforge.net/"。

(8) Reportlab:Reportlab 用于绘制 PDF 报表,官网地址为"http://www.reportlab.com/software/opensource/"。

(9) Vpython:VPython 是 Visual Python 的简写,它是一个 Python 3D 绘图模块。其官网地址为"http://www.vpython.org/index.html"。

13.2　基于 tkinter 的图形绘制

13.2.1　基于 tkinter 的画布绘图概述

Canvas(画布)是一个长方形的区域,用于图形绘制或复杂的图形界面布局,可以在画布上绘制图形、文字,放置各种组件和框架。

Canvas 组件对象有下列方法(绘制函数),用于绘制各种图形对象。

- create_arc():绘制圆弧。
- create_bitmap():绘制位图。
- create_image():绘制位图图像。
- create_line():绘制线。
- create_oval():绘制椭圆。
- create_polygon():绘制多边形。
- create_rectangle():绘制矩形。
- create_text():绘制文本。
- create_window():绘制子窗口。

13.2.2　创建画布对象

画布的左上角为坐标原点(0,0),右下角为画布的大小(x,y)。创建画布对象的语法格式如下:

```
c = tkinter.Canvas(parent, option = value, … )
```

其中,parent 为父组件,默认无;option 为选项,通过如下命令可以列举其选项。

```
>>> from tkinter import *
>>> c = Canvas(); c.keys()
['background', 'bd', 'bg', 'borderwidth', 'closeenough', 'confine', 'cursor', 'height',
'highlightbackground','highlightcolor', 'highlightthickness', 'insertbackground', 'insertborderwidth', '
insertofftime', 'insertontime', 'insertwidth', 'offset', 'relief', 'scrollregion', 'selectbackground', '
selectborderwidth', 'selectforeground', 'state', 'takefocus', 'width', 'xscrollcommand', '
xscrollincrement', 'yscrollcommand', 'yscrollincrement']
```

【例 13.1】　创建一个大小为 200×100、背景为黄色的画布(canvas.py)。程序运行效果如图 13-1 所示。

```
from tkinter import *          # 导入 tkinter 模块的所有内容
root = Tk()                    # 创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'yellow', width = 200, height = 100);
                               # 创建大小为 200×100、背景为黄色的画布
c.pack()                       # 调用组件的 pack()方法,调整其显示
                               # 位置和大小
```

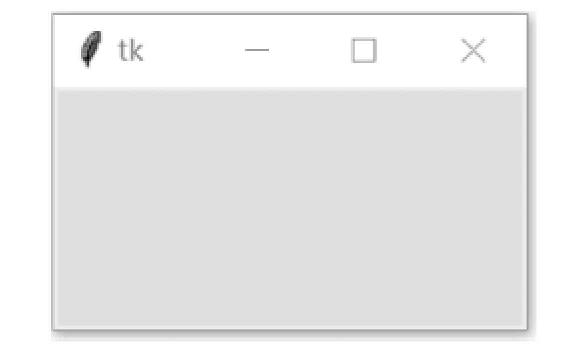


图 13-1　创建画布

13.2.3　绘制矩形

在 Canvas 对象 c 上绘制矩形的方法如下:

```
id = c.create_rectangle(x0, y0, x1, y1, option, … )
```

其中,(x0, y0)是左上角的坐标;(x1, y1)是右下角的坐标。

【例 13.2】　矩形绘制示例(create_rectangle.py)。程序运行效果如图 13-2 所示。



图 13-2　绘制矩形

```
from tkinter import *                    #导入 tkinter 模块的所有内容
root = Tk()                              #创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width=130, height=70); c.pack()
                                         #创建并显示 Canvas
c.create_rectangle(10,10,60,60,fill='red')
                                         #用红色填充矩形
c.create_rectangle(70,10,120,60,fill='red',outline='blue',width=5)
                                         #用蓝色边框、红色填充矩形,边框宽度为 5
```

13.2.4　绘制椭圆

在 Canvas 对象 c 上绘制椭圆的方法如下:

```
id = c.create_oval(x0, y0, x1, y1, option, …)
```

其中,(x0, y0)是左上角的坐标;(x1, y1)是右下角的坐标。

【例 13.3】　椭圆绘制示例(create_oval.py)。程序运行效果如图 13-3 所示。

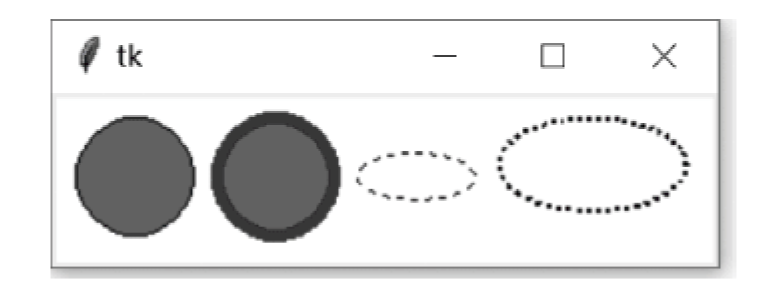


图 13-3　绘制椭圆

```
from tkinter import *                                           #导入 tkinter 模块的所有内容
root = Tk()                                                     #创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width=280, height=70); c.pack()   #创建并显示 Canvas
c.create_oval(10,10,60,60,fill='red')                           #用红色填充椭圆
c.create_oval(70,10,120,60,fill='red',outline='blue',width=5)   #红色填充、蓝色边框、宽度为 5 的
                                                                #椭圆
c.create_oval(130,25,180,45,dash=(4,))                          #虚线椭圆
c.create_oval(190,10,270,50,dash=(4,),width=2)                  #宽度为 2 的虚线椭圆
```

13.2.5　绘制圆弧

在 Canvas 对象 c 上绘制圆弧的方法如下:

```
id = c.create_arc(x0, y0, x1, y1, option, …)
```

(x0, y0)是左上角的坐标;(x1, y1)是右下角的坐标;option 为选项,其中,选项 start(开始角度,默认为 0)和 extent(圆弧角度,从 start 开始逆时针转,默认为 90°)决定圆弧的角度范围,选项 style 用于设置圆弧的样式,取值为 tk.PIESLICE(默认值)、tk.CHORD 和 tk.ARC,对应的形状样式如图 13-4 所示。

【例 13.4】　圆弧绘制示例(create_arc.py)。程序运行效果如图 13-5 所示。

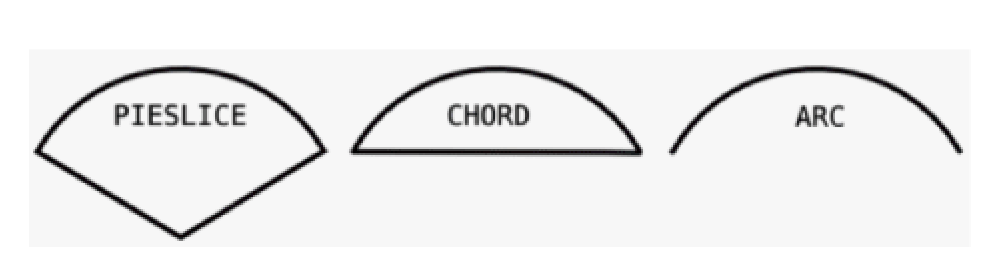


图 13-4　圆弧形状样式



图 13-5　绘制圆弧

```
from tkinter import *                                          # 导入 tkinter 模块的所有内容
root = Tk()                                                    # 创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width=250, height=70); c.pack()  # 创建并显示 Canvas
c.create_arc(10,10,60,60, style=PIESLICE)                      # 绘制 PIESLICE 样式圆弧
c.create_arc(70,10,120,60, style=CHORD)                        # 绘制 CHORD 样式圆弧
c.create_arc(130,10,180,60, style=ARC)                         # 绘制 ARC 样式圆弧
for i in range(0, 360, 60):                                    # 绘制菊花瓣蓝色边框、红色填
                                                               # 充的图形
    c.create_arc(190,10,240,60,fill='red',outline='blue',start=i,extent=30)
```

13.2.6 绘制线条

在 Canvas 对象 c 上绘制线条(直线或折线)的方法如下:

id = c.create_line(x0, y0, x1, y1, …, xn, yn, option, …)

其中,(x0, y0),(x1, y1),…,(xn, yn)是线条上各个点的坐标。

【例 13.5】 线条绘制示例(create_line.py)。程序运行效果如图 13-6 所示。



图 13-6 绘制线条

```
from tkinter import *                                          # 导入 tkinter 模块的所有内容
root = Tk()                                                    # 创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width=250, height=70); c.pack()  # 创建并显示 Canvas
c.create_line(10,10,60,60,arrow=BOTH,arrowshape=(3,5,4))       # 双向箭头
c.create_line(70,10,95,10,120,60)                              # 折线
c.create_line(130,10,180,10,130,60,180,60,width=10,arrow=BOTH)        # Z 字形双向箭头
c.create_line(190,10,240,10,190,60,240,60,width=10,joinstyle=MITER) # Z 字形
```

13.2.7 绘制多边形

在 Canvas 对象 c 上绘制多边形的方法如下:

id = c.create_polygon(x0, y0, x1, y1, …, option, …)

其中,(x0, y0),(x1, y1),…,(xn, yn)是多边形各个顶点的坐标。

【例 13.6】 多边形绘制示例(create_polygon.py)。程序运行效果如图 13-7 所示。

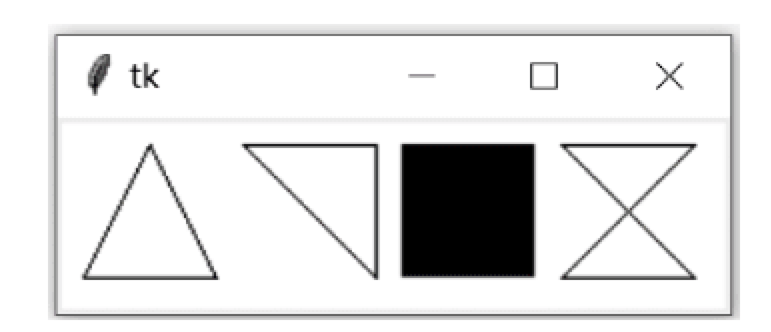


图 13-7 绘制多边形

```
from tkinter import *                                          # 导入 tkinter 模块的所有内容
root = Tk()                                                    # 创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width=250, height=70); c.pack()  # 创建并显示 Canvas
c.create_polygon(35,10,10,60,60,60,fill='white',outline='black')      # 黑边等腰三角形
```

```
c.create_polygon(70,10,120,10,120,60,fill = 'white',outline = 'black')  #黑边直角三角形
c.create_polygon(130,10,180,10,180,60,130,60)                            #黑色填充的正方形
c.create_polygon(190,10,240,10,190,60,240,60,fill = 'white',outline = 'black')   #对顶三角形
```

13.2.8 绘制字符串

在 Canvas 对象 c 上绘制字符串的方法如下：

id = c.create_text(x, y, option, …)

(x, y)是字符串放置的中心坐标；option 为选项，包括 activefill、activestipple、anchor、disabledfill、disabledstipple、fill、font、justify、offset、state、stipple、tags、text、width，其中，text 用于指定要绘制的字符串，font 用于指定字体，justify 用于指定对齐方式。

13.2.9 应用举例：绘制函数图形

【例 13.7】 字符串和图形绘制(sin. py)：绘制给定函数 y＝sin(x)的图形。程序运行效果如图 13-8 所示。

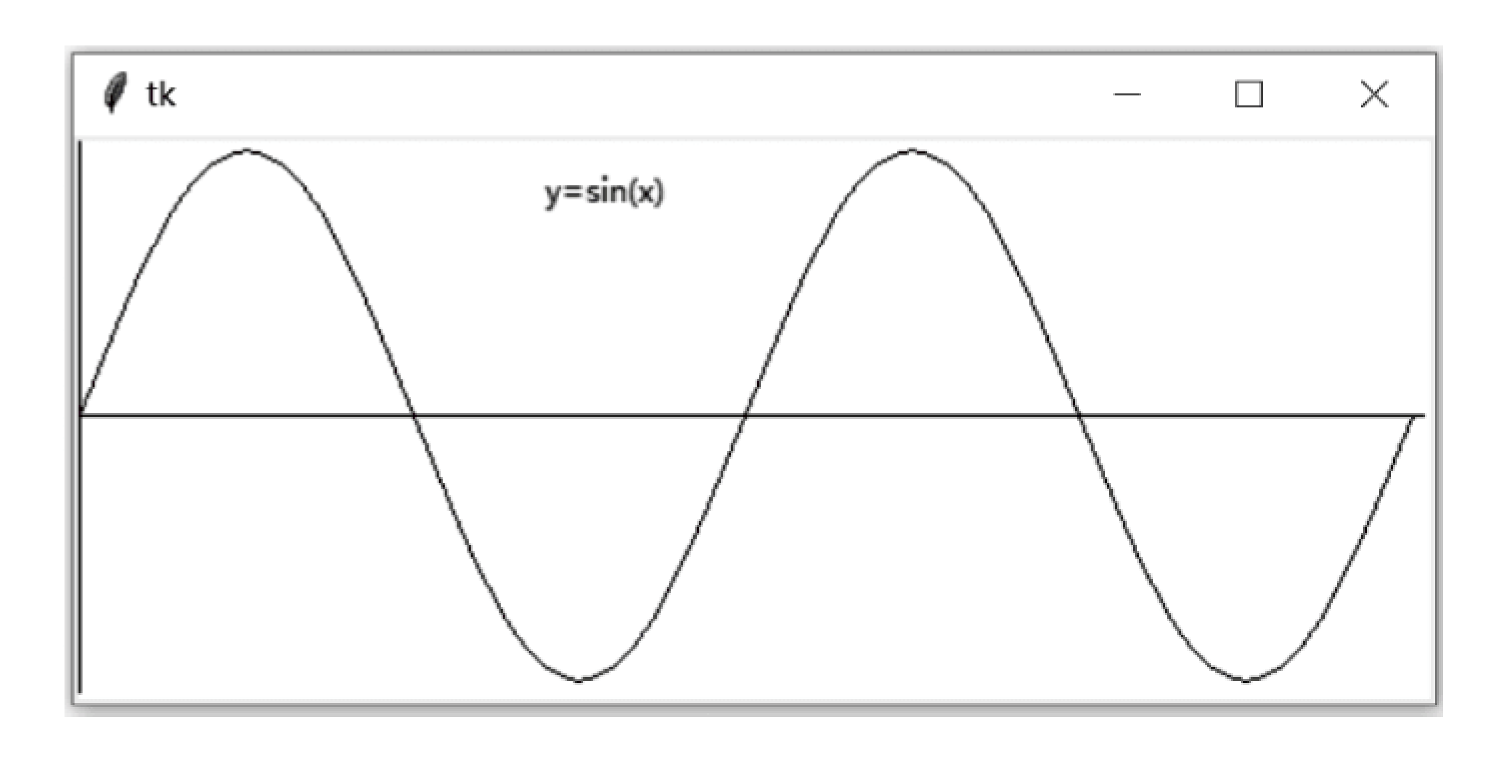


图 13-8 绘制函数

```
from tkinter import *                                    #导入 tkinter 模块的所有内容
import math                                              #导入 math 模块
WIDTH = 510; HEIGHT = 210                                #画布的宽度、高度
ORIGIN_X = 2; ORIGIN_Y = HEIGHT / 2                      #原点(X = 2,Y = 窗体左边中心)
SCALE_X = 40; SCALE_Y = 100                              #X 轴、Y 轴的缩放倍数
END_ARC = 360 * 2                                        #函数图形画多长(两个周期)
ox = 0; oy = 0; x = 0; y = 0                             #坐标初始值
arc = 0                                                  #弧度
root = Tk()                                              #创建一个 Tk 根窗口组件 root
c = Canvas(root,bg = 'white', width = WIDTH, height = HEIGHT);c.pack()  #创建并显示 Canvas
c.create_text(200, 20, text = 'y = sin(x)')                             #绘制文字
c.create_line(0, ORIGIN_Y, WIDTH, ORIGIN_Y)                             #绘制 Y 轴
c.create_line(ORIGIN_X, 0, ORIGIN_X, HEIGHT)                            #绘制 X 轴
for i in range(0, END_ARC + 1, 10):                                     #绘制函数图形
    arc = math.pi * i * 2 / 360
    x = ORIGIN_X + arc * SCALE_X
    y = ORIGIN_Y - math.sin(arc) * SCALE_Y
    c.create_line(ox, oy, x, y)
    ox = x; oy = y
```

13.3　基于 turtle 模块的海龟绘图

13.3.1　海龟绘图概述

所谓海龟绘图，即假定一只海龟(海龟带着一支钢笔)在一个屏幕上来回移动，当它移动时会绘制直线。海龟可以沿直线移动指定的距离，也可以旋转一个指定的角度。

通过编写代码可以控制海龟移动，从而绘制出图形。使用海龟作图不仅能够通过简单的代码创建出令人印象深刻的视觉效果，而且可以跟随海龟动态查看程序代码如何影响海龟的移动和绘制，从而有助于理解代码的逻辑。

13.3.2　turtle 模块概述

Python 标准库中的 turtle 模块基于 tkinter 的画布实现了海龟绘图的功能。使用 turtle 模块绘图一般遵循如下步骤。

(1) 导入 turtle 模块。

```
from turtle import *                          #将 turtle 模块中的所有方法导入
```

(2) 创建海龟对象(turtle 模块同时实现了函数模式，故也可以不创建海龟对象，直接调用函数绘图)。

```
p = Turtle()                                  #创建海龟对象
```

(3) 设置海龟的绘图属性(画笔的属性，颜色、线条的宽度)。

```
pensize(width)/width(width)                   #绘制图形时的宽度
color(colorstring)                            #绘制图形时的画笔颜色和填充颜色
pencolor(colorstring)                         #绘制图形的画笔颜色
fillcolor(colorstring)                        #绘制图形的填充颜色
```

(4) 控制和操作海龟绘图。

```
pendown()/pd()/down()                         #移动时绘制图形，默认为绘制
penup()/pu()/up()                             #移动时不绘制图形
forward(distance)/fd(distance)                #向前移动 distance 指定的距离
backward(distance)/bk(distance)/back(distance) #向后移动 distance 指定的距离
right(angle)/rt(angle)                        #向右旋转 angle 指定的角度
left(angle)/lt(angle)                         #向左旋转 angle 指定的角度
goto(x,y)/setpos(x,y)/setposition(x,y)        #将画笔移动到坐标为(x,y)的位置
dot(size = None, * color)                     #绘制指定大小的圆点
circle(radius, extent = None, steps = None)   #绘制指定大小的圆
write(arg, move = False, align = 'left', font = ('Arial', 8, 'normal'))   #绘制文本
stamp()                                       #复制当前图形
speed(speed)                                  #画笔绘制的速度([0,10]的整数)
showturtle()/st()                             #显示海龟
hideturtle()/ht()                             #隐藏海龟
clear()                                       #清除海龟绘制的图形
reset()                                       #清除海龟绘制的图形并重置海龟属性
```

13.3.3 绘制正方形

【例 13.8】 使用海龟绘图绘制一个正方形（square.py）。程序运行效果如图 13-9 所示。

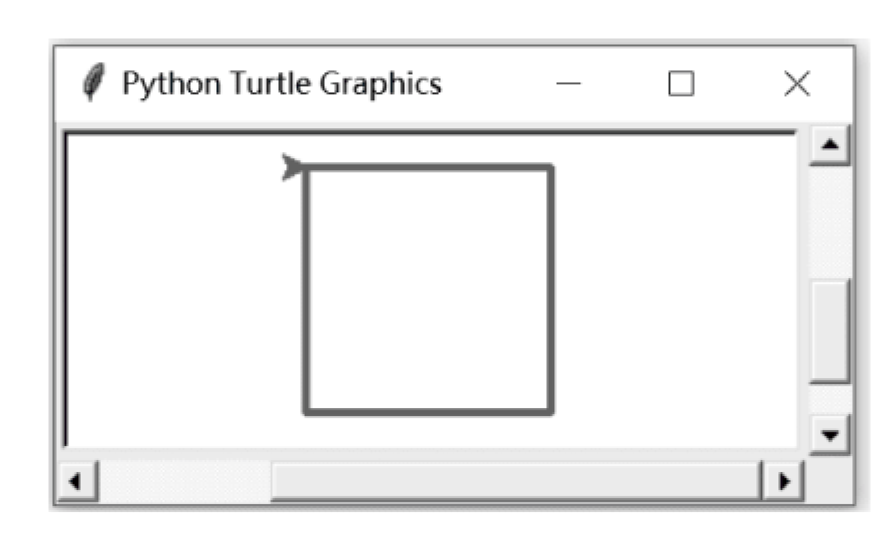


图 13-9 使用海龟绘图绘制一个正方形

```
import turtle          # 导入 turtle 模块
p = turtle.Turtle()    # 创建海龟对象
p.color("red")         # 设置绘制时画笔的颜色
p.pensize(3)           # 定义绘制时画笔的线条宽度
turtle.speed(1)        # 定义绘图的速度("slowest"或者
                       # 1 均表示最慢)
p.goto(0,0)            # 移动海龟到坐标原点(0,0)
for i in range(4):     # 绘出正方形的 4 条边
    p.forward(100)     # 向前移动 100
    p.right(90)        # 向右旋转 90°
```

说明：在使用海龟绘图时，其原点（0，0）位于画布区域的中央位置。

13.3.4 绘制多边形

【例 13.9】 使用海龟绘图绘制三角形、正方形、正五边形、……、正十边形等多边形（polygon.py）。程序运行效果如图 13-10 所示。

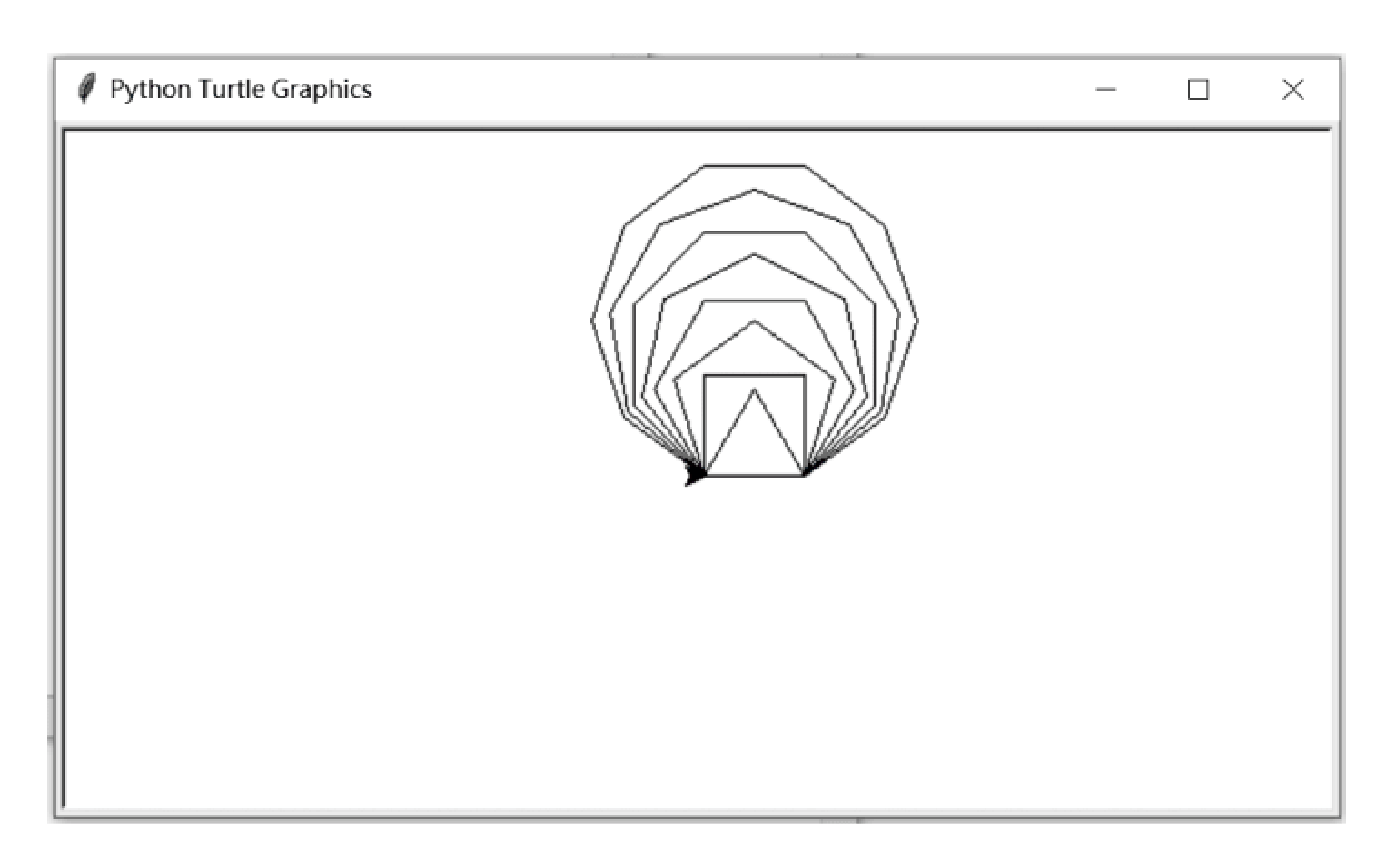


图 13-10 使用海龟绘图绘制各种多边形

```
import turtle                           # 导入 turtle 模块
def draw_polygon(sides, side_len):      # 绘制指定边长长度的多边形
    for i in range(sides):
        turtle.forward(side_len)        # 绘制边长
        turtle.left(360.0/sides)        # 旋转角度
def main():
    for i in range(3,11):               # 绘制三角形、正方形、正五边形、……、正十边形
        step = 50                       # 边长(海龟步长)为 50
        draw_polygon(i, step)           # 绘制多边形
if __name__ == '__main__': main()
```

说明：本例直接调用 turtle 模块的海龟绘图函数，没有创建海龟对象。

13.3.5 绘制圆形螺旋

【例 13.10】 使用海龟绘图分别绘制红、蓝、绿、黄 4 种颜色的圆形螺旋(spiral.py)。程序运行效果如图 13-11 所示。

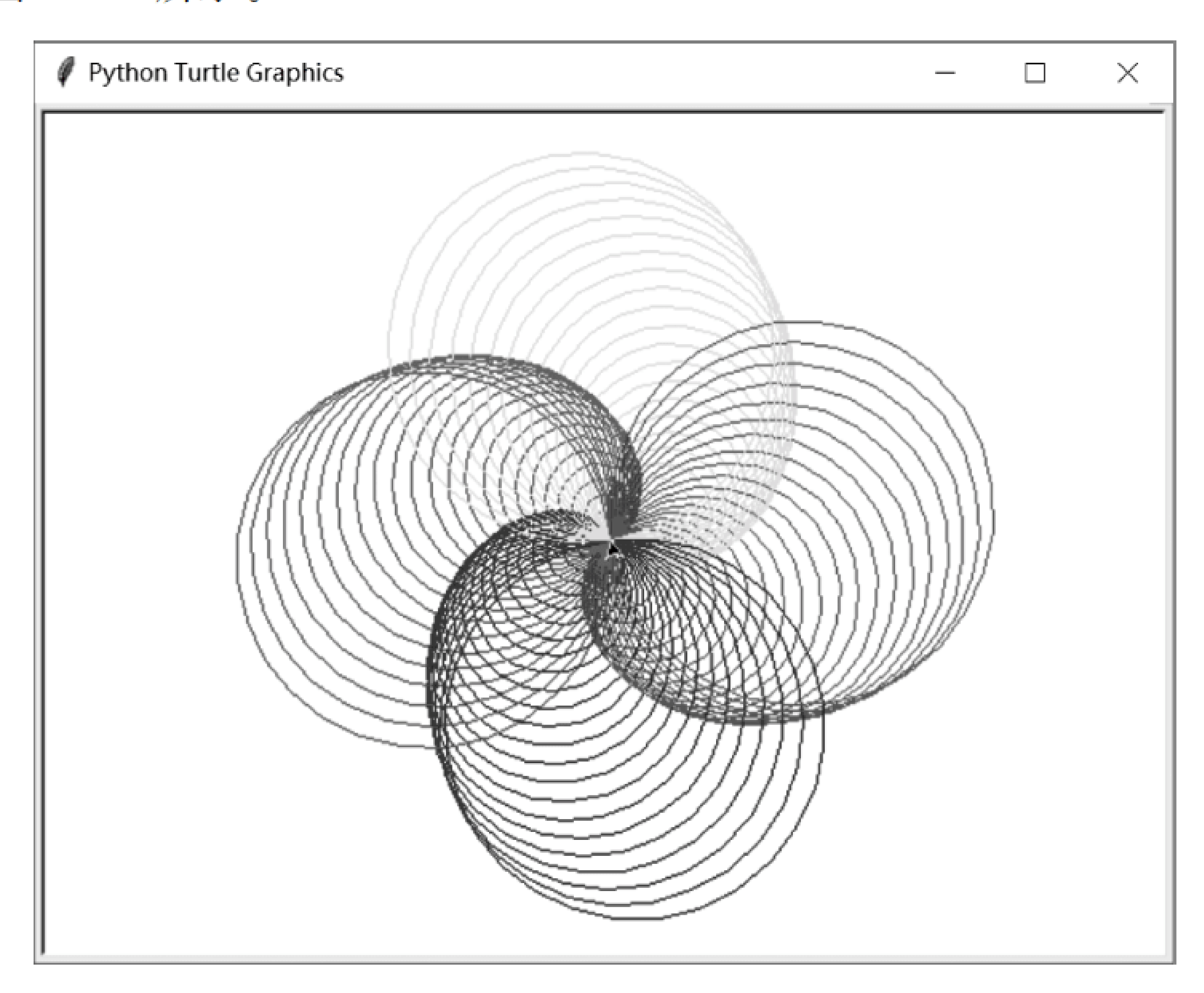


图 13-11 使用海龟绘图绘制 4 种颜色的圆形螺旋

```
import turtle                                        # 导入 turtle 模块
p = turtle.Turtle()                                  # 创建海龟对象
p.speed(0)                                           # 定义绘图的速度("fastest"或者 0 均表示最快)
colors = ["red", "blue", "green", "yellow"]          # 红、蓝、绿、黄 4 种颜色
for i in range(100):                                 # i = 0～99
    p.pencolor(colors[i % 4])                        # 设置画笔颜色(红、蓝、绿或黄)
    p.circle(i)                                      # 画圆
    p.left(91)                                       # 向左旋转 91 度
```

13.3.6 递归图形

分形(Fractal)概念由法国数学家曼德布罗在 1975 年提出,用于形容局部与整体相似的形状。分形图可以使用简单的递归绘图方案实现,从而产生复杂的图像。分形图可以模拟自然界中的树、蕨类、云等。

【例 13.11】 科赫曲线(Koch curve)的绘制(Koch.py)。

n 阶科赫曲线的递归绘制算法如下。

(1) 基本情况(当 n=0 时):绘制一条直线。

(2) 递归步骤(当 n≥1 时):绘制一条阶数为 n−1 的科赫曲线;向左旋转 60°,绘制第 2 条阶数为 n−1 的科赫曲线;向右旋转 120°,绘制第 3 条阶数为 n−1 的科赫曲线;向左旋转 60°,绘制第 4 条阶数为 n−1 的科赫曲线。

n 阶科赫曲线的绘制线条的长度是 n−1 阶科赫曲线长度的 1/3。

3 阶科赫曲线的绘制结果如图 13-12 所示。

```
import sys                                # 导入 sys 模块
import turtle                             # 导入 turtle 模块
def koch(t, order, size):
```

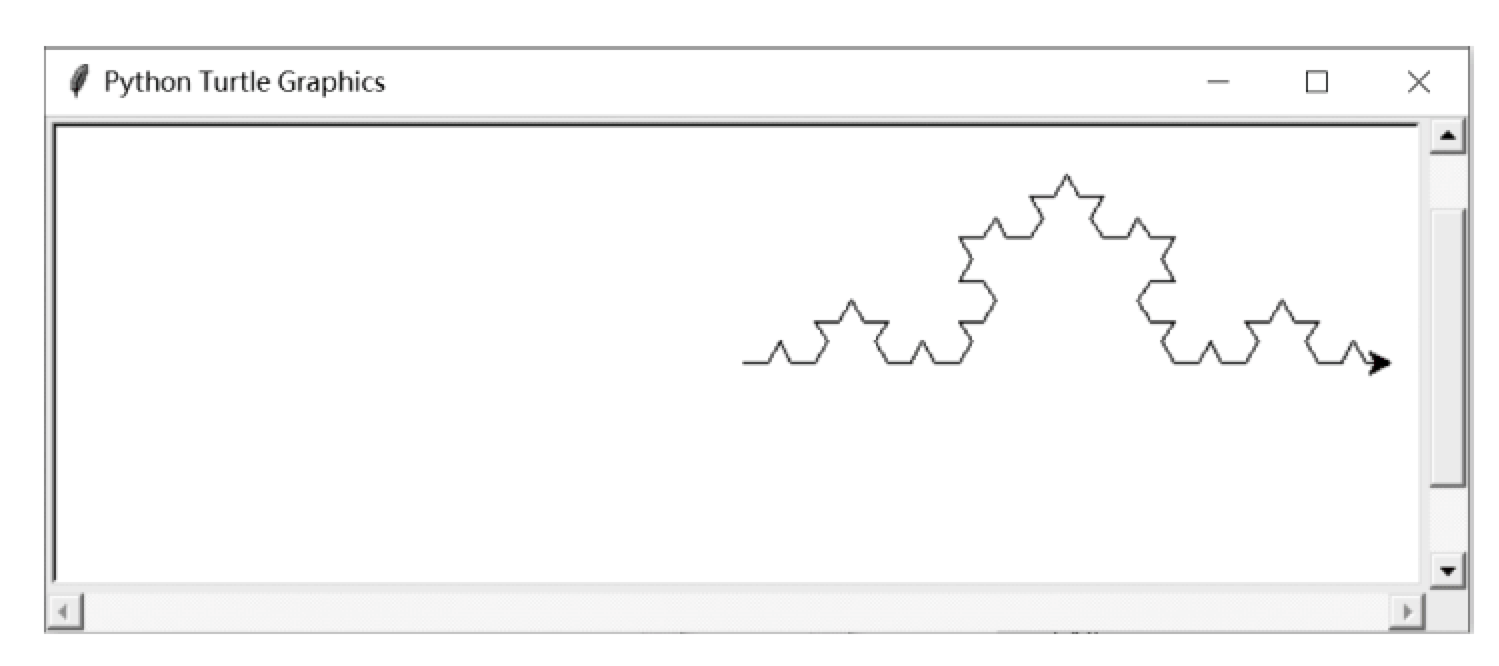


图 13-12 使用海龟绘图绘制 n 阶科赫曲线

```
        if order == 0:                  #当 n 等于 0 时，绘制一条直线
            t.forward(size)
        else:                           #否则，递归绘制 n 阶科赫曲线
            koch(t, order - 1, size/3)  #递归绘制一条阶数为 n-1 的科赫曲线，长度为 1/3
            t.left(60.0)                #向左旋转 60°
            koch(t, order - 1, size/3)  #递归绘制一条阶数为 n-1 的科赫曲线，长度为 1/3
            t.right(120.0)              #t.left(-120.0) #向右旋转 120°(或者向左旋转-120°)
            koch(t, order - 1, size/3)  #递归绘制一条阶数为 n-1 的科赫曲线，长度为 1/3
            t.left(60.0)                #向左旋转 60°
            koch(t, order - 1, size/3)  #递归绘制一条阶数为 n-1 的科赫曲线，长度为 1/3
def main():
        n = int(sys.argv[1])            #n 阶科赫曲线
        step = 300                      #步长
        p = turtle.Turtle()             #创建海龟对象
        koch(p, n, step)                #绘制 n 阶科赫曲线
if __name__ == '__main__': main()
```

13.3.7 海龟绘图的应用实例

【例 13.12】 使用 Python Turtle Demo 学习海龟绘图的应用实例。

（1）运行 Python 内置集成开发环境 IDLE。

（2）运行 Python Turtle Demo。选择 IDLE 中的 Help | Turtle Demo 命令，打开 Turtle Demo 源代码编辑器，如图 13-13 所示。

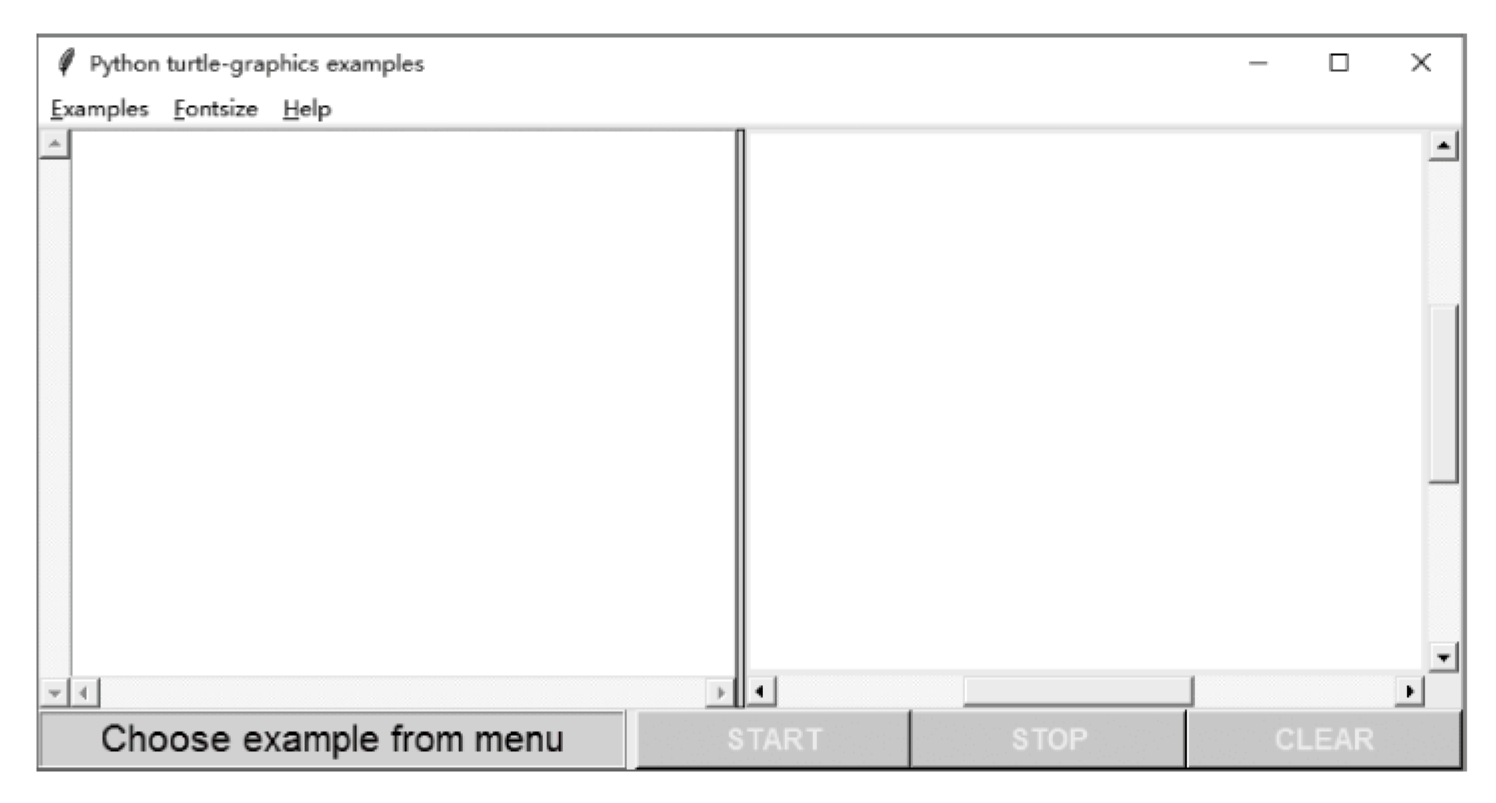


图 13-13 Turtle Demo 源代码编辑器

（3）查看海龟绘图示例 clock。在 Turtle Demo 中选择 Examples | clock 命令，打开 clock 示例，左边窗格中显示源代码；单击窗口下部的 START 按钮，可以查看程序运行结果，如图 13-14 所示。

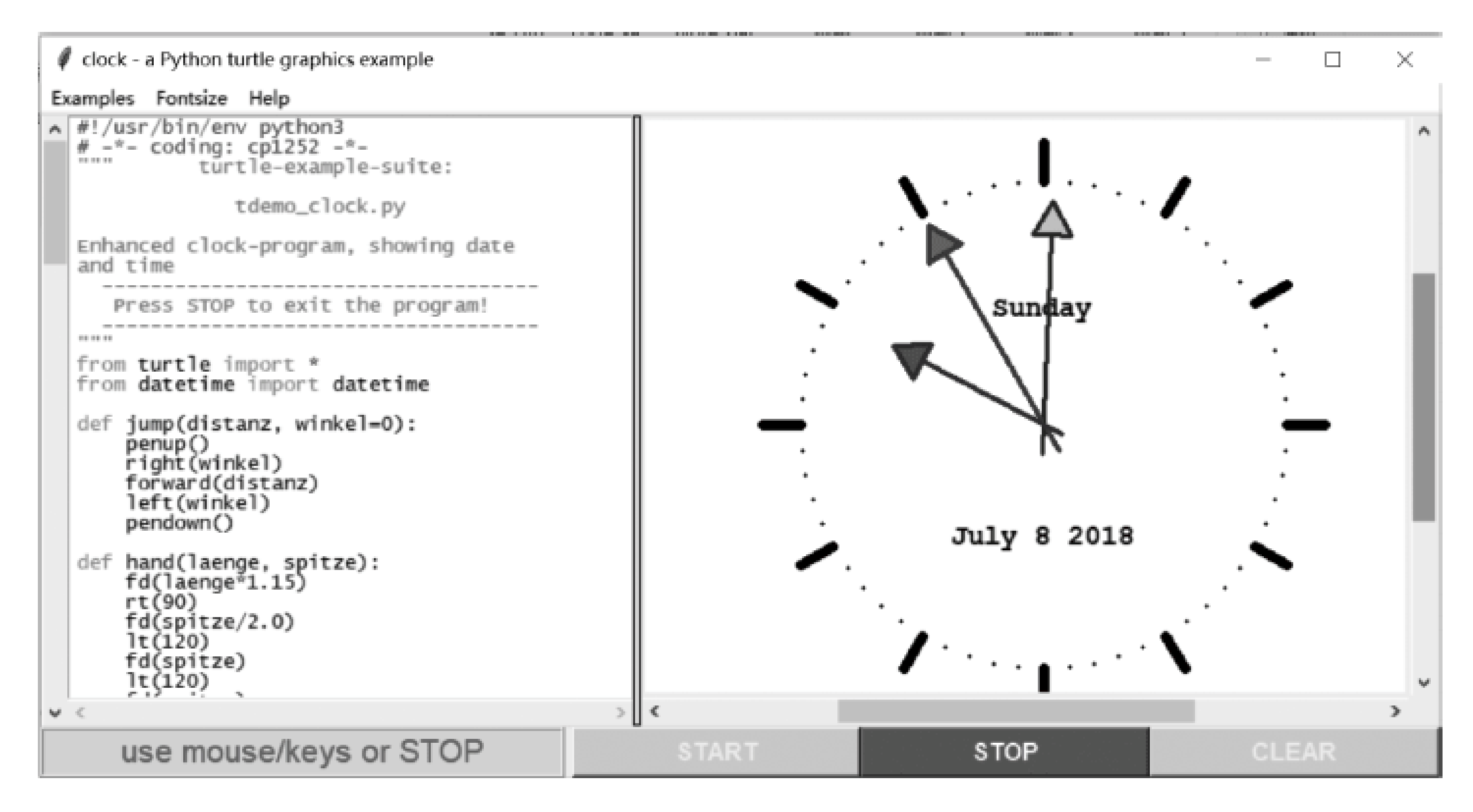


图 13-14　查看海龟绘图示例 clock

（4）查看其他海龟绘图示例。参照步骤(3)，查看其他示例。

13.4　基于 Matplotlib 模块的绘图

13.4.1　Matplotlib 模块概述

Matplotlib 是 Python 最著名的绘图库之一，提供了一整套和 MATLAB 相似的命令 API，既适合交互式地进行制图，也可以作为绘图控件方便地嵌入 GUI 应用程序中。

Matplotlib 的 pyplot 子库提供了和 MATLAB 类似的绘图 API，方便用户快速绘制 2D 图表，包括直方图、饼图、散点图等。

Matplotlib 配合 NumPy 模块使用，可以实现科学计算结果的可视化显示。

Matplotlib 的文档相当完备，其官网的 Gallery 页面(http://matplotlib.org/gallery.html)中包括各种类型图形的示例图，如图 13-15 所示。选择需要绘制某种类型的图形，可以查看相应的源代码，复制、修改源代码以满足实际需求。其官网的 examples 页面(https://matplotlib.org/examples/index.html)中包含了大量的示例程序。



图 13-15　Matplotlib 官网的 Gallery 页面

13.4.2　安装 Matplotlib 模块

Matplotlib 的官网地址是“http://matplotlib.org/”，用户可以直接从官网下载安装 Matplotlib 模块。

建议用户使用 Python 包管理工具 pip 安装 Matplotlib 模块,详细步骤请参照本书第 1 章的相关内容。

13.4.3 使用 Matplotlib 模块绘图概述

使用 Matplotlib 模块绘图主要用 Matplotlib.pyplot 工具包。

Matplotlib 是一套面向对象的绘图库,其所绘制图表中的每个绘图元素(例如线条、文字、刻度等)都是对象。

为了实现快速绘图,Matplotlib 的子模块 pyplot 提供了与 MATLAB 类似的绘图 API,封装了复杂的绘图对象结构。用户只需要调用 pyplot 模块所提供的函数就可以实现快速绘图,并设置图表的各种细节。

Matplotlib.pyplot 是命令行式函数的集合,每一个函数都对图像做了修改,例如创建图形、在图像上创建画图区域、在画图区域上画线、在线上标注等。

【例 13.13】 使用 plot()函数画图(linecurve.py):绘制 X 轴坐标值为 0、1、2、3、4,所对应 Y 轴坐标值为 1、2、5、6、8 的折线图。

```
import matplotlib.pyplot as plt            # 导入 Matplotlib 模块中的子模块 pyplot
plt.plot([1, 2, 5, 6, 8])
plt.ylabel('some numbers')                 # 为 Y 轴加注释
plt.show()
```

程序运行效果如图 13-16 所示。

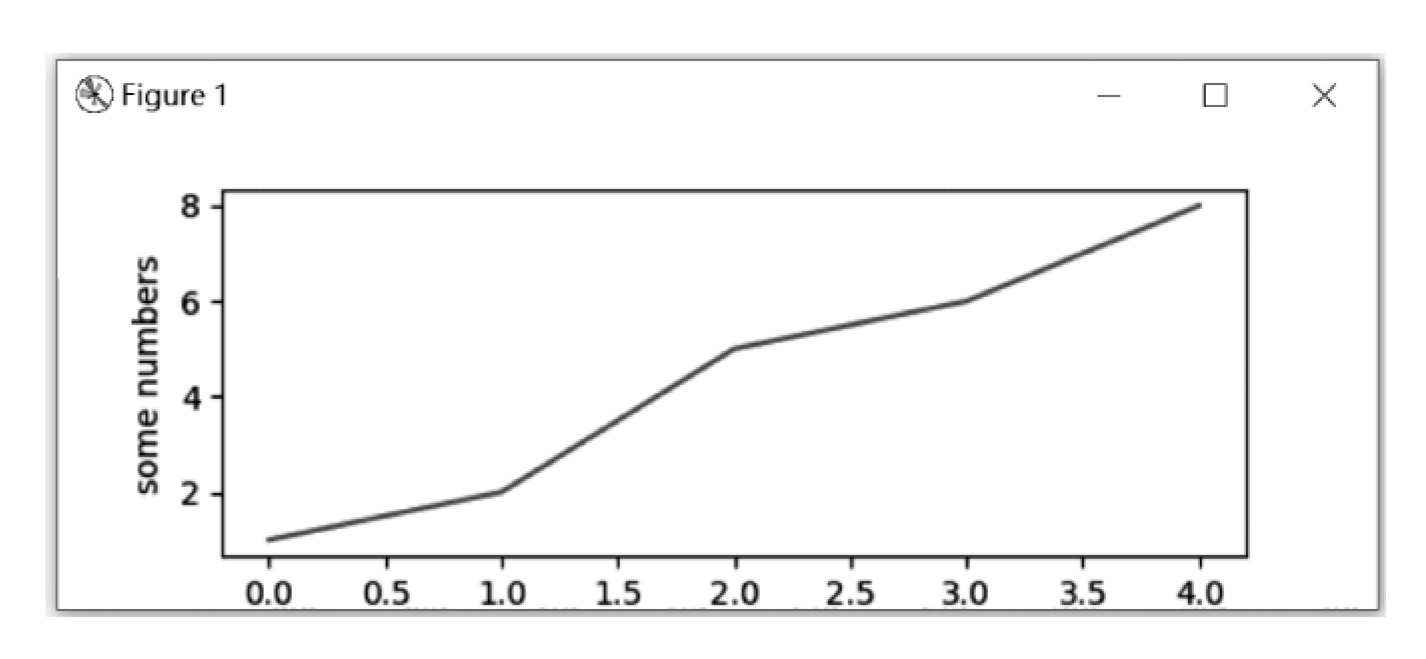


图 13-16 使用 plot()函数画折线图

13.4.4 绘制函数曲线

【例 13.14】 使用 Matplotlib 模块绘制 y=sin(x)的函数曲线(sine.py)。

```
import matplotlib.pyplot as plt            # 导入 Matplotlib 模块中的子模块 pyplot
import math                                # 导入 math 模块
x = [2 * math.pi * i/100 for i in range(100)]
y = [math.sin(i) for i in x]
plt.plot(x, y)
plt.show()
```

程序运行效果如图 13-17 所示。

13.4.5 绘制多个图形

pyplot 和 MATLAB 一样,支持绘制多个子图。figure()命令用于指定图形,subplot()命令用于指定一个坐标系。例如,figure(1)(默认)指定图形 1,subplot(211)指定子图 1(上、下

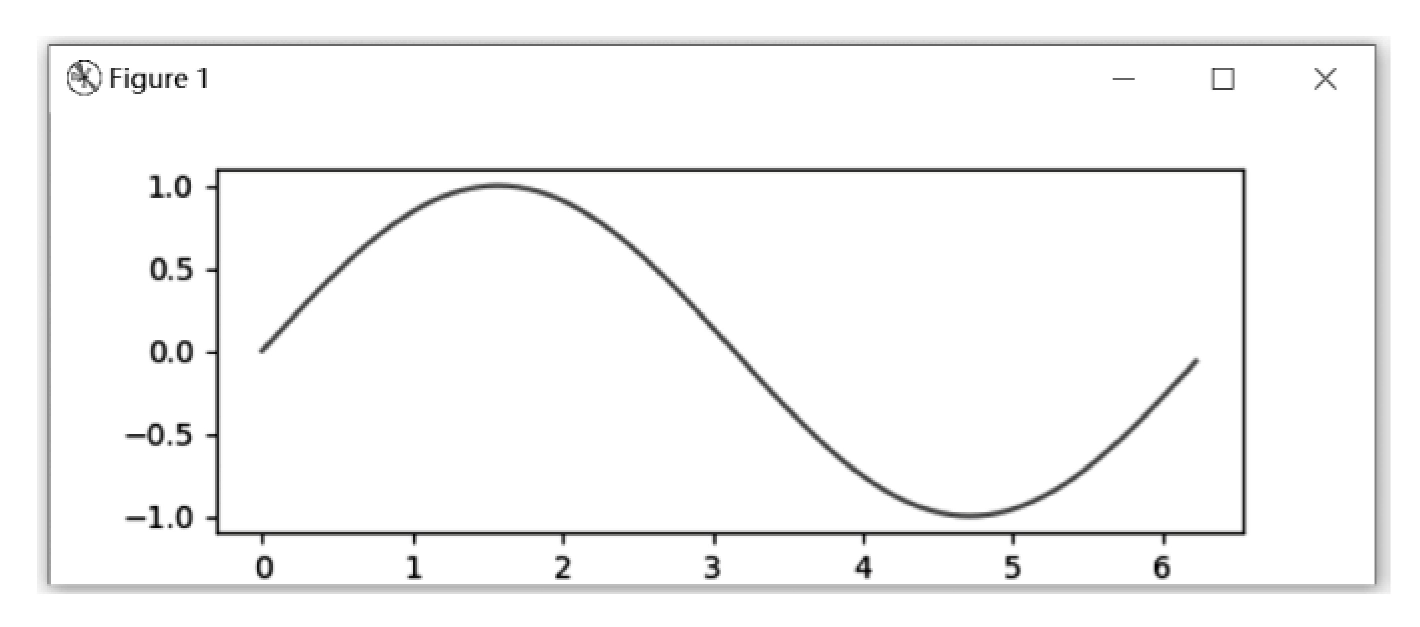


图 13-17 使用 Matplotlib 模块绘制 y=sin(x)的函数曲线

两个子图的第 1 幅图像),subplot(212)指定子图 2(上、下两个子图的第 2 幅图像)。在指定了子图后,所有绘图命令都是针对当前图像和当前子图。

【例 13.15】 绘制多个子图示例(multifig.py):利用 NumPy 模块和 Matplotlib.pyplot 工具包绘制 $y=e^{-t}\times\cos(2\pi x)$以及 $y=\cos(2\pi x)$的函数曲线。

```
import numpy as np                       # 导入 NumPy 模块
import matplotlib.pyplot as plt          # 导入 Matplotlib 模块中的子模块 pyplot
def f(t):
  return np.exp(-t) * np.cos(2 * np.pi * t)
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(212)
plt.plot(t2, np.cos(2 * np.pi * t2), 'r-- ')
plt.show()
```

程序运行效果如图 13-18 所示。

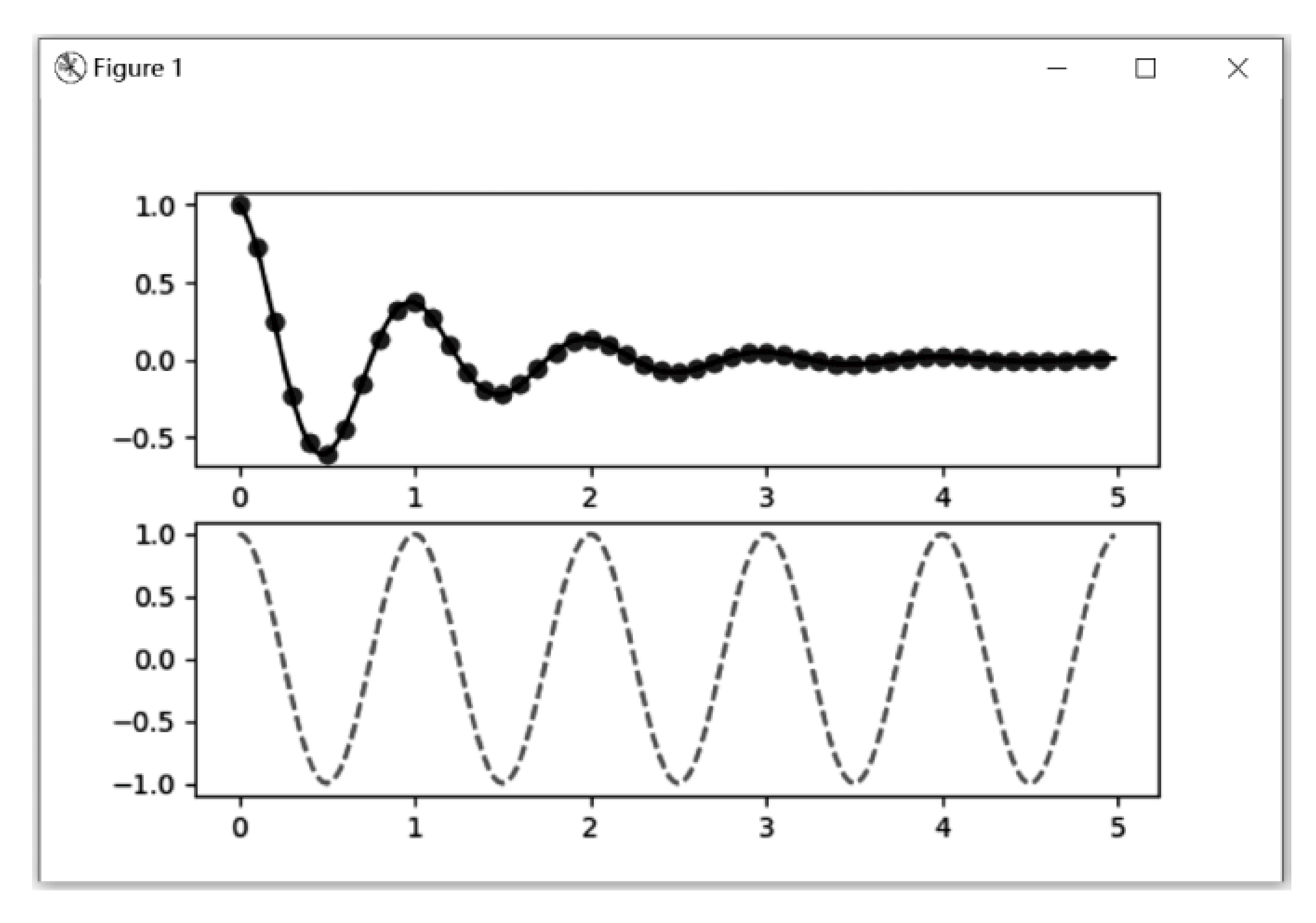


图 13-18 使用 Matplotlib 模块绘制多个子图

说明:该程序中使用了 NumPy 模块,生成用于绘制的图形数据。

13.4.6 绘制直方图

直方图是由一系列高度不等的纵向条纹或线段表示数据分布的统计报告图,使用 Matplotlib.pyplot 的 hist()函数可以方便、快捷地绘制直方图。

【例 13.16】 使用 Matplotlib.pyplot 的 hist()函数绘制直方图示例(histfig.py):随机生成满足 mu 为 100、sigma 为 20 的正态分布的 10 万个智商数据,并绘制其直方图。

```
import numpy as np                           #导入 NumPy 模块
import matplotlib.pyplot as plt              #导入 Matplotlib 模块中的子模块 pyplot
#随机生成满足 mu 为 100、sigma 为 20 的正态分布的 10 万个智商数据
mu, sigma = 100, 20
x = mu + sigma * np.random.randn(100000)
#绘制直方图
plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)
#绘制坐标等信息
plt.xlabel('IQ');plt.ylabel('Probability')
plt.title('Histogram of IQ')
#设置坐标和网格
plt.axis([40, 180, 0, 0.03])
plt.grid(True)
plt.show()
```

程序运行效果如图 13-19 所示。

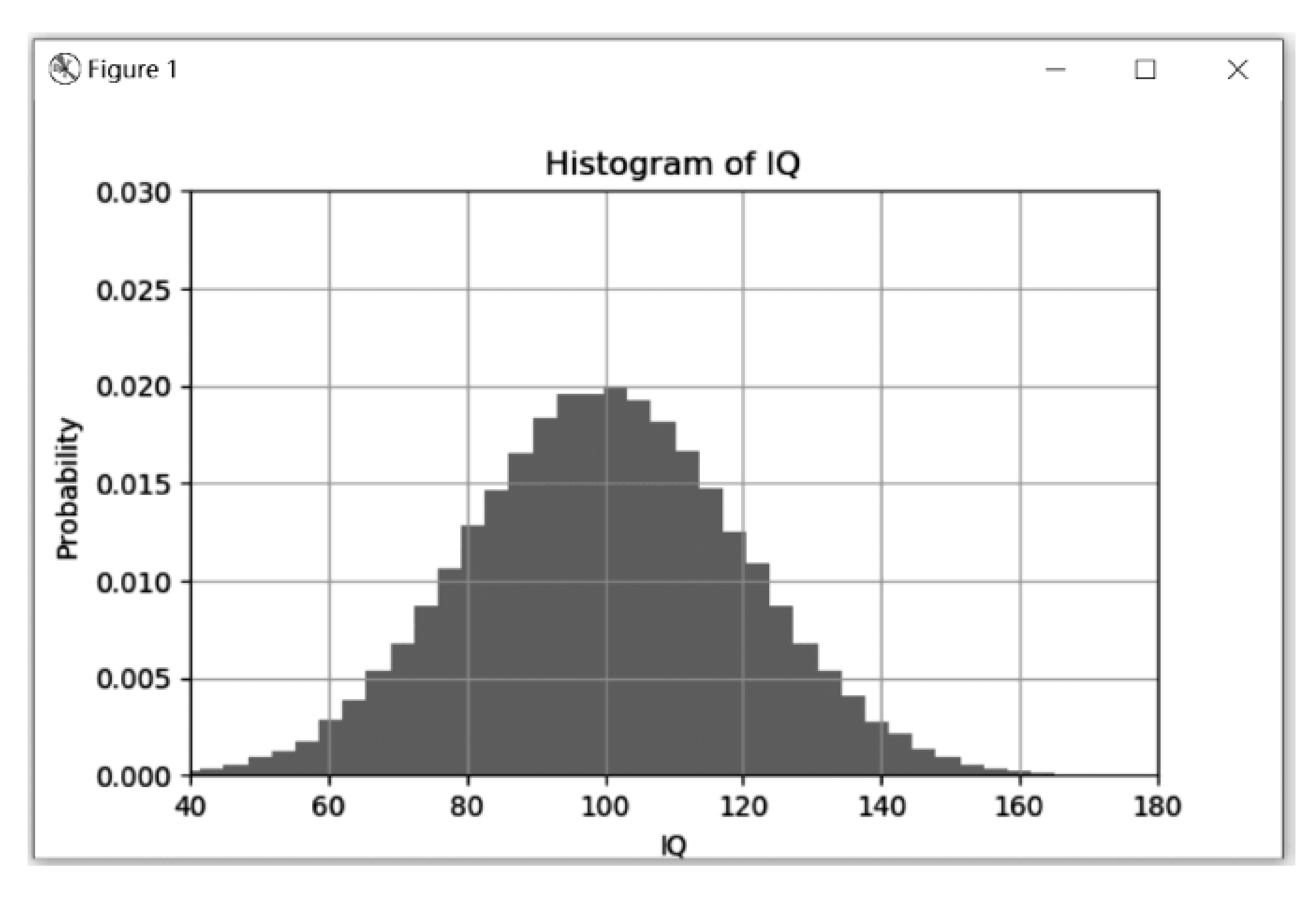


图 13-19 使用 Matplotlib.pyplot 的 hist()函数绘制直方图

13.5 复 习 题

1. 在 Python 标准库中包括图形绘制相关模块,其中,模块________用于画布绘图,模块________用于海龟绘图。

2. ________是一个长方形的区域,用于图形绘制或复杂的图形界面布局,可以在其上绘制图形、文字,放置各种组件和框架。

3. 画布的________为坐标原点(0,0),________为画布的大小(x,y)。

4. Matplotlib 是 Python 最著名的绘图库之一,其________子库(子模块)提供了和 MATLAB 类似的绘图 API,方便用户快速绘制 2D 图表,包括直方图、饼图、散点图等。

13.6 上 机 实 践

1. 完成本章中的例 13.1～例 13.16,熟悉 Python 语言中基于 tkinter 模块的图形绘制、基于 turtle 模块的图形绘制以及基于 Matplotlib 模块的图形绘制。

2. 参照例 13.2 利用 Canvas 组件创建绘制矩形的程序，尝试改变矩形边框颜色以及填充颜色。

3. 参照例 13.3 利用 Canvas 组件创建绘制椭圆的程序，尝试改变椭圆边框样式、边框颜色以及填充颜色。

4. 参照例 13.4 利用 Canvas 组件创建绘制圆弧的程序，尝试改变圆弧样式、边框颜色以及填充颜色。

5. 参照例 13.5 利用 Canvas 组件创建绘制线条的程序，尝试改变线条样式和颜色。

6. 参照例 13.6 利用 Canvas 组件创建绘制多边形的程序，尝试改变多边形的形状、线条样式和填充颜色。

7. 参照例 13.7 利用 Canvas 组件创建绘制字符串和图形的程序，绘制函数 y＝cos(x)的图形，程序运行效果如图 13-20 所示。

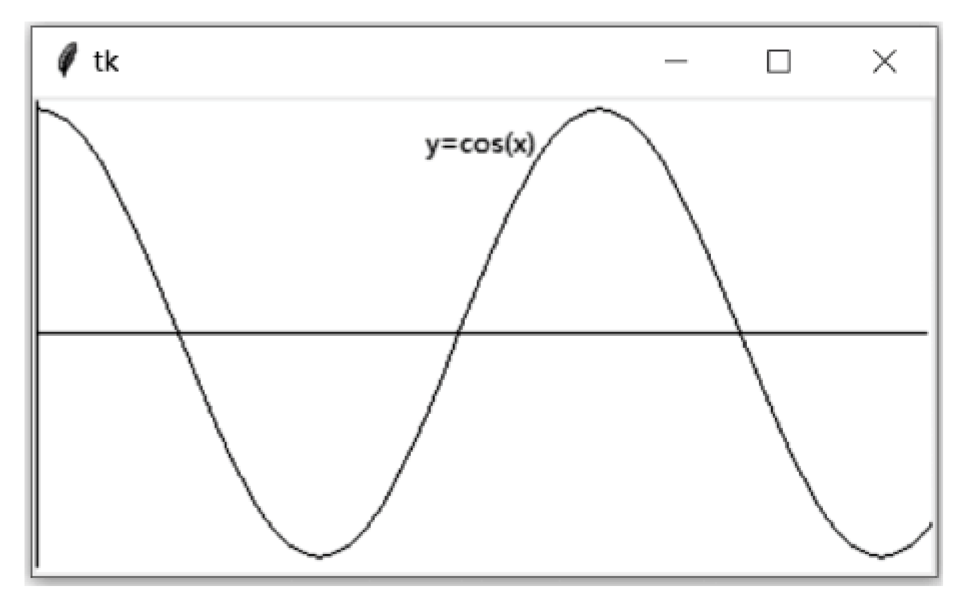


图 13-20 函数 y＝cos(x)的程序运行效果

8. 参照例 13.8 和例 13.9 编程，使用海龟绘图在同一水平线上分别绘制一个红色的三角形、绿色的正方形、蓝色的正五边形并且书写说明文字。其中，红色的三角形从原点开始绘制，3 个图形间相隔 100 点。最后在与三角形相隔 200 点处打印“绘制完成！”的字样，Arial 字体、20 字号。程序运行效果如图 13-21 所示。



图 13-21 绘制红色三角形、绿色正方形和蓝色正五边形以及书写文字

9. 参照例 13.13，使用 plot()函数画图，绘制 X 轴坐标值为 0、1、2、3、4，所对应 Y 轴坐标值为 y＝6x＋5 的直线图。程序运行效果如图 13-22 所示。

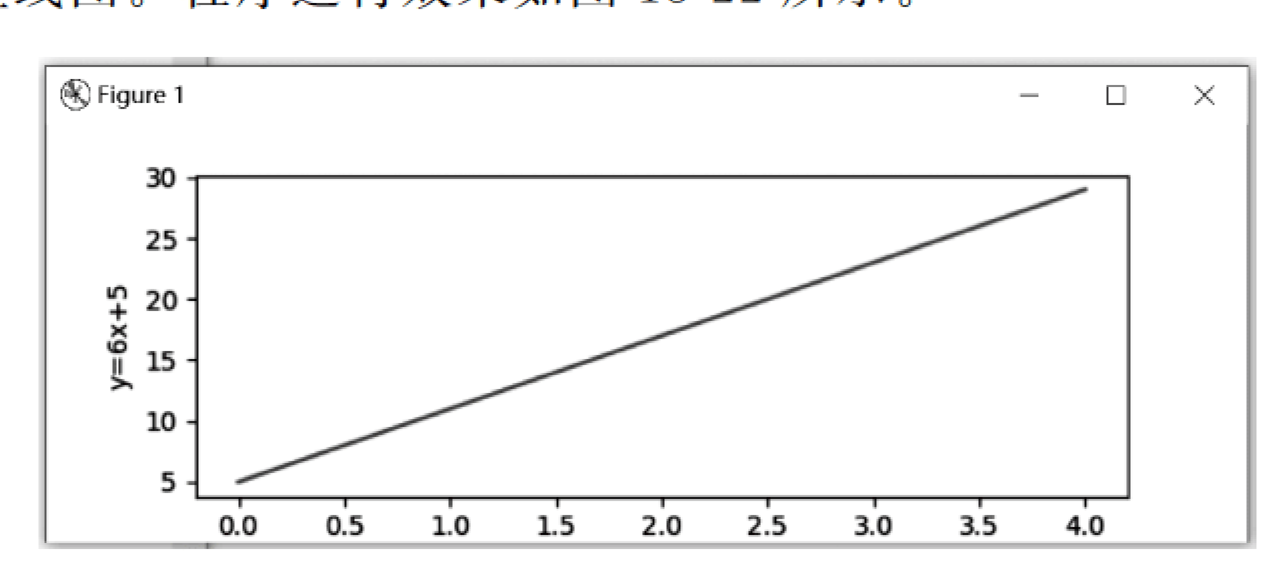


图 13-22 绘制 y＝6x＋5 的直线图

10. 参照例 13.14,使用 Matplotlib 模块绘制 y=cos(x)的函数曲线。程序运行效果如图 13-23 所示。

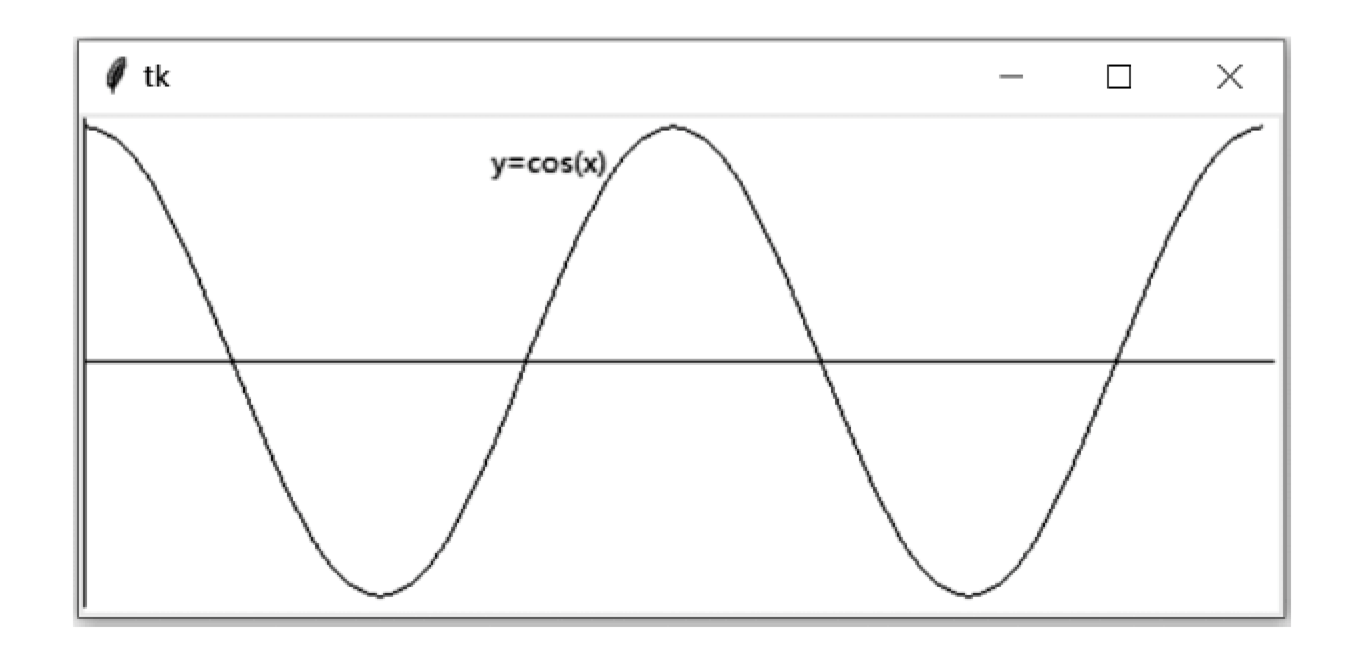


图 13-23 使用 Matplotlib 模块绘制 y=cos(x)的函数曲线

11. 参照例 13.15,利用 NumPy 模块和 Matplotlib.pyplot 工具包绘制 $y=e^{-t}\times\sin(2\pi x)$以及 $y=\sin(2\pi x)$的函数曲线。程序运行效果如图 13-24 所示。

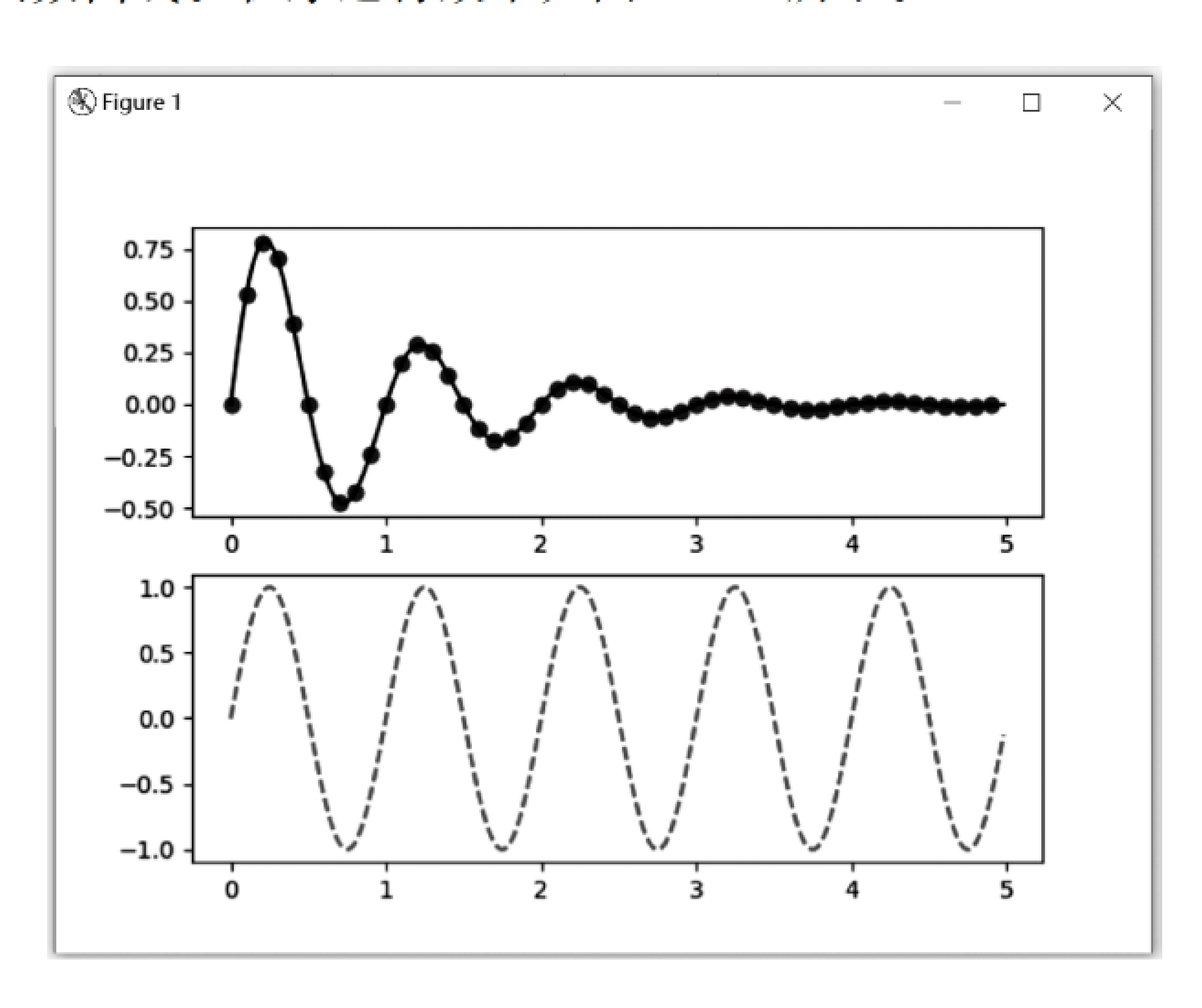


图 13-24 使用 Matplotlib 模块绘制多个子图

13.7 案例研究:汉诺塔问题求解动画

本章案例研究是一个基于 turtle 的汉诺塔问题求解动画的设计和实现,用来帮助读者进一步深入了解递归和 turtle 图形。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

第 14 章

数值日期和时间处理


视频讲解

Python 提供了丰富的数据类型和模块函数，用于程序设计中的数值处理、日期和时间处理。

14.1 相关模块概述

14.1.1 数值处理的相关模块

1. Python 标准库中的数值处理相关模块

- numbers 模块：数值抽象类，包含类 Complex、Real、Rational、Integral。
- math 模块：数学函数。
- cmath 模块：复数运算数学函数。
- decimal 模块：高精度数值运算。
- fractions 模块：分数运算模块。
- random 模块：随机数模块。

2. 数值运算模块 NumPy

Python 扩展模块 NumPy 提供了更高效的数值处理功能。NumPy 模块主要提供数组和矩阵处理功能，还包括高级功能，例如傅里叶变换等。NumPy 的官网地址为"http://www.numpy.org"。

3. 科学计算模块 SciPy

Python 扩展模块 SciPy 提供了用于科学计算的功能。SciPy 模块包括统计、优化、整合、线性代数、傅里叶变换、信号和图像处理、常微分方程求解器等功能。SciPy 的官网地址为"http://www.scipy.org"。

14.1.2 日期和时间处理的相关模块

Python 标准库中的 datetime 模块包含各种用于日期和时间处理的类；calendar 模块包含用于处理日历的函数和类；time 模块包含用于处理时间的函数。

14.2 math 模块和数学函数

14.2.1 math 模块的 API

Python 标准模块 math 中提供了许多常用的数学函数，包括三角函数、对数函数和其他通用数学函数。math 模块中的函数不支持复数，复数函数位于 cmath 模块。

math 模块包含的常量和函数，其 API 如表 14-1 所示。其中的三角函数以弧度为单位。假设该表中的示例基于“from math import ＊”。

表 14-1（1） math 的常量和函数（一）：常量

名　　称	说　　明	示　　例	结　　果
e	数学常量 e	e	2.718281828459045
pi	数学常量 pi	pi	3.141592653589793

表 14-1（2） math 的常量和函数（二）：数值运算和表示

名　　称	说　　明	示　　例	结　　果
ceil(x)	返回≥x 的最小整数	ceil(1.2)，ceil(－1.6)	2，－1
copysign(x，y)	返回符号为 y 的 x 的值	copysign(1.0，－0.0)	－1.0
fabs(x)	返回 x 的绝对值	fabs(－1.2)	1.2
factorial(x)	返回正整数 x 的阶乘	factorial(10)	3628800
floor(x)	返回≤x 的最大整数	floor(1.8)，floor(－2.1)	1，－3
fmod(x，y)	返回 x % y 的值	fmod(5，3)	2.0
frexp(x)	返回(m，e)，使得： x == m ＊ 2 ** e	frexp(1024)	(0.5，11)
fsum(iterable)	返回序列之和，浮点数的计算结果比 sum 更精确	fsum([.1，.1，.1，.1，.1，.1，.1，.1，.1，.1]) sum([.1，.1，.1，.1，.1，.1，.1，.1，.1，.1])	1.0 0.9999999999999999
isfinite(x)	判断 x 是否为有限值	isfinite(float('Infinity')) isfinite(float('－Infinity')) isfinite(float('NaN')) isfinite(float('12.3'))	False False False True
isinf(x)	判断 x 是否为无穷大	isinf(float('Infinity')) isinf(12.3)	True False
isnan(x)	判断 x 是否为非数值	isnan(float('NaN')) isnan(12.3)	True False
ldexp(x，i)	返回 x ＊ (2 ** i) 它是 frexp(x)的反函数	ldexp(2，10)	2048.0
modf(x)	返回 x 的小数和整数部分，结果为元组	modf(－12.3)	(－0.3000000000000007，－12.0)
trunc(x)	将 x 截为最接近 0 的整数	trunc(1.2) trunc(－2.8)	1 －2

表 14-1（3） math 的常量和函数（三）：幂和对数运算

名　　称	说　　明	示　　例	结　　果
exp(x)	返回 e ** x	exp(5)	148.4131591025766
expm1(x)	返回 e ** x － 1 比 exp(x)－1 精确	expm1(1e－5) exp(1e－5) －1	1.0000050000166667e－05 1.0000050000069649e－05
log(x) log(x，base)	返回 $\log_e x$ 返回 $\log_{base} x$	log(e) log(e，2)	1.0 1.4426950408889634

续表

名　称	说　明	示　例	结　果
log1p(x)	返回 log(1+x)	log1p(1) log(2)	0.6931471805599453 0.6931471805599453
log2(x)	返回 $\log_2 x$	log2(e)	1.4426950408889634
log10(x)	返回 $\log_{10} x$	log10(100)	2.0
pow(x, y)	返回 x^y，即 x ** y	pow(2,8)	256.0

表 14-1(4)　math 的常量和函数(四)：三角函数

名　称	说　明	示　例	结　果
acos(x)	返回 x 的反余弦	acos(1)	0.0
asin(x)	返回 x 的反正弦	asin(1)	1.5707963267948966
atan(x)	返回 x 的反正切	atan(1)	0.7853981633974483
atan2(y, x)	返回 x 的 atan(y/x)	atan2(1, 2)	0.4636476090008061
cos(x)	返回 x 的余弦	cos(2 * pi)	1.0
hypot(x, y)	返回 sqrt(x * x + y * y)，即欧几里得距离	hypot(3,4)	5.0
sin(x)	返回 x 的正弦	sin(pi/2)	1.0
tan(x)	返回 x 的正切	tan(pi/4)	0.9999999999999999

表 14-1(5)　math 的常量和函数(五)：双曲线函数

名　称	说　明	示　例	结　果
acosh(x)	返回 x 的双曲线反余弦	acosh(1)	0.0
asinh(x)	返回 x 的双曲线反正弦	asinh(1)	0.8813735870195429
atanh(x)	返回 x 的双曲线反正切	atanh(0.1)	0.1003353477310756
cosh(x)	返回 x 的双曲线余弦	cosh(1)	1.5430806348152437
sinh(x)	返回 x 的双曲线正弦	sinh(0.1)	0.10016675001984403
tanh(x)	返回 x 的双曲线正切	tanh(0.1)	0.09966799462495582

表 14-1(6)　math 的常量和函数(六)：角度弧度转换函数

名　称	说　明	示　例	结　果
degrees(x)	将 x 从弧度转换为角度	degrees(pi)	180.0
radians(x)	将 x 从角度转换为弧度	radians(90)	1.5707963267948966

表 14-1(7)　math 的常量和函数(七)：其他函数

名　称	说　明	示　例	结　果
erf(x)	返回 x 的误差函数	(1.0+erf(1/sqrt(2.0)))/2.0	0.841344746068543
erfc(x)	返回 1.0 −erf(x)	erfc(1)	0.157299207050285
gamma(x)	返回 x 的伽马函数	gamma(50)	6.082818640342675e+62
lgamma(x)	返回 x 的伽马函数的绝对值的自然对数	lgamma(50)	144.5657439463449

说明：如果 x 不是 float 数据类型，则 ceil(x)、floor(x)、trunc(x)等同于 x 的对象方法 x.__ceil__()、x.__floor__()、x.__trunc__()。

14.2.2 math 模块应用举例

【例 14.1】 数学函数的使用示例(math_test.py)：输入三条边长，如果可以构成三角形，则求三角形的面积、周长，以及某边长所对应的高、最长边长、最短边长；否则报错“不能构成三角形”。

```
import math
#三角形三边 a、b、c 必须满足:三条边长均大于零,并且任意两边之和大于第三边
a = int(input("请输入边长 a:"))
b = int(input("请输入边长 b:"))
c = int(input("请输入边长 c:"))
if (a > 0 and b > 0 and c > 0 and a + b > c and a + c > b and b + c > a):
    p = (a + b + c) / 2                                    #周长的一半
    area = math.sqrt(p * (p - a) * (p - b) * (p - c))      #面积
    perimeter = a + b + c                                  #周长
    height_a = 2 * area / a                                #边长 a 所对应的高
    max_side = max(a, b, c)                                #最长边长
    min_side = min(a, b, c)                                #最短边长
    print("三角形的三条边为:{0}、{1}和{2}".format(a, b, c))
    print("三角形的面积为:{0:.2f}".format(area))
    print("三角形的周长为:{0:.2f}".format(perimeter))
    print("边长 A 对应的高为:{0:.2f}".format(height_a))
    print("三角形的最长的边为:{0:.2f}".format(max_side))
    print("三角形的最短的边为:{0:.2f}".format(min_side))
else:
    print("三条边:{0}、{1}和{2},不能构成三角形".format(a, b, c))
```

程序运行结果如下。

```
请输入边长 a:3
请输入边长 b:4
请输入边长 c:5
三角形的三条边为:3、4 和 5
三角形的面积为:6.00
三角形的周长为:12.00
边长 A 对应的高为:4.00
三角形的最长的边为:5.00
三角形的最短的边为:3.00
```

【例 14.2】 数学函数的使用示例(quadratic.py)：求一元二次方程 $x^2+bx+c=0$ 的实数解。其中，系数 b 和 c 由命令行参数所确定。

```
import math
import sys
b = float(sys.argv[1])
c = float(sys.argv[2])
discriminant = b * b - 4.0 * c
if discriminant >= 0:
    d = math.sqrt(discriminant)
    print("x1 =",(-b + d) / 2.0)
    print("x2 =",(-b - d) / 2.0)
else:
    print("此方程无实数解")
```

程序运行结果如图 14-1 所示。

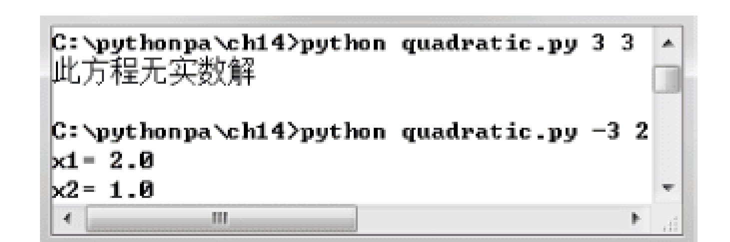


图 14-1　求解一元二次方程的实数根

14.3　cmath 模块和复数数学函数

Python 标准模块 cmath 中提供了许多用于复数运算的函数。

cmath 模块包含的常量和函数，其 API 如表 14-2 所示。假设该表中的示例基于“from cmath import *”。

注意：cmath 模块中函数的参数可以为整数、浮点数、复数，或者其他可以自动转换为复数或浮点数的对象，即具有__complex__()或__float__()方法的对象。

表 14-2(1)　cmath 的常量和函数(一)：常量

名　称	说　明	示　例	结　果
e	数学常量 e	e	2.718281828459045
pi	数学常量 pi	pi	3.141592653589793

表 14-2(2)　cmath 的常量和函数(二)：转换函数(笛卡儿坐标和极坐标)

名　称	说　明	示　例	结　果
phase(x)	返回 math.atan2(x.imag, x.real)	phase(complex(−1.0, 0.0))	3.141592653589793
polar(x)	返回(abs(x), phase(x))，即(r, phi)	polar(3+4j)	(5.0, 0.9272952180016122)
rect(r, phi)	返回(r, phi)对应的复数，即 r * (cos(phi) + sin(phi) * 1j)	rect(1,pi/4)	(0.7071067811865476+0.7071067811865476j)

表 14-2(3)　cmath 的常量和函数(三)：幂和对数运算

名　称	说　明	示　例	结　果
exp(x)	返回 e ** x	exp(3+4j)	(−13.128783081462158−15.200784463067954j)
log10(x)	返回 $\log_{10} x$	log10(3+4j)	(0.6989700043360187+0.4027191962733731j)
sqrt(x)	返回 x 的平方根	sqrt(3+4j)	(2+1j)

表 14-2(4)　cmath 的常量和函数(四)：三角函数

名　称	说　明	示　例	结　果
acos(x)	返回 x 的反余弦	acos(3+4j)	(0.9368124611557198−2.305509031243477j)
asin(x)	返回 x 的反正弦	asin(3+4j)	(0.6339838656391766+2.305509031243477j)
atan(x)	返回 x 的反正切	atan(3+4j)	(1.4483069952314644+0.1589971916799918j)
cos(x)	返回 x 的余弦	cos(2 * pi)	(1+0j)
sin(x)	返回 x 的正弦	sin(pi/2)	(1+0j)
tan(x)	返回 x 的正切	tan(pi/4)	(0.9999999999999999+0j)

表 14-2(5) cmath 的常量和函数(五)：双曲线函数

名　　称	说　　明	示　　例	结　　果
acosh(x)	返回 x 的双曲线反余弦	acosh(1)	0j
asinh(x)	返回 x 的双曲线反正弦	asinh(1)	(0.881373587019543+0j)
atanh(x)	返回 x 的双曲线反正切	atanh(0.1)	(0.10033534773107558+0j)
cosh(x)	返回 x 的双曲线余弦	cosh(1)	(1.5430806348152437+0j)
sinh(x)	返回 x 的双曲线正弦	sinh(0.1)	(0.10016675001984403+0j)
tanh(x)	返回 x 的双曲线正切	tanh(0.1)	(0.09966799462495582+0j)

表 14-2(6) cmath 的常量和函数(六)：判别函数

名　　称	说　　明	示　　例	结　　果
isfinite(x)	判断 x.real 和 x.imag 是否都为有限值	isfinite(3+4j)	True
isinf(x)	判断 x.real 和 x.imag 是否其一为无穷大	isinf(3+4j)	False
isnan(x)	判断 x.real 和 x.imag 是否其一为非数值	isnan(3+4j)	False

14.4 random 模块和随机函数

random 模块包含各种伪随机数生成函数，以及各种根据概率分布生成随机数的函数。该模块中的大部分函数都基于 random()函数，该函数使用 Mersenne Twister 生成器在[0.0，1.0)范围内生成一致分布的随机值。

14.4.1 种子和随机状态

使用 random 模块函数 seed()可以设置伪随机数生成器的种子。其基本形式如下：

```
random.seed(a = None, version = 2)
```

其中 a 为种子。当没有指定 a 时使用系统时间，如果 a 为整数，则直接使用。当 a 不为整数且 version=2 时 a 转换为整数，否则使用 a 的哈希值。

同一种子，每次运行产生的随机数相同(故称之为伪随机数)。例如：

```
>>> import random
>>> random.seed(1)                                          #设置种子为 1
>>> for i in range(5): print(random.randint(1,5),end = ',')     #输出:2,5,1,3,1,
>>> for i in range(5): print(random.randint(1,5),end = ',')     #输出:4,4,4,4,2,
>>> random.seed(1)                                          #重新设置种子为 1,结果重复
>>> for i in range(5): print(random.randint(1,5),end = ',')     #输出:2,5,1,3,1,
>>> random.seed(10)                                         #重新设置种子为 10
>>> for i in range(5): print(random.randint(1,5),end = ',')     #输出:5,1,4,4,5,
```

14.4.2 随机整数

random 模块中用于生成随机整数的函数如表 14-3 所示。假设该表中的示例基于“from random import *”。

表 14-3　random 模块中的随机整数函数

名　　称	说　　明	示　　例	结果(随机)
randrange(stop)	返回随机整数 N,N 属于序列[0, stop)	for i in range(10): print(randrange(10),end=',')	8,6,7,4,0,9,1,5,4,9
randrange(start, stop[, step])	返回随机整数 N,N 属于序列[start, stop, step)	for i in range(10): print(randrange(1,5),end=',')	3,4,1,2,3,4,4,4,1,1
randint(a, b)	返回随机整数 N,使得 a≤N≤b,即 randrange(a, b+1)	for i in range(10): print(randint(1,5),end=',')	1,2,5,5,1,1,1,2,3,4
getrandbits(k)	返回随机整数 N,使得 N 的位(bit)长为 k	for i in range(10): print(getrandbits (2),end=',')	0,0,1,2,1,3,1,1,1,1

【例 14.3】 猜数游戏(guess. py):首先随机产生一个 1～100 的整数,请用户猜测具体是哪个数,即不断从标准输入读取用户的猜测值,并根据猜测值给出提示信息"太大""太小"或"正确!"。

```
import random
secret = random.randrange(1, 101)
guess = 0
while guess != secret:
    guess = int(input("请猜测一个 100 之内的数:"))
    if (guess < secret): print('太小')
    elif (guess > secret): print('太大')
    else: print('正确!')
```

程序运行结果(随机数由系统随机产生,因此每次的运行结果有所不同)如下。

```
请猜测一个 100 之内的数:50
太大
请猜测一个 100 之内的数:25
太大
请猜测一个 100 之内的数:13
太大
请猜测一个 100 之内的数:8
太大
请猜测一个 100 之内的数:4
太大
请猜测一个 100 之内的数:2
正确!
```

14.4.3　随机序列

random 模块中用于从序列中随机抽取数据的函数如表 14-4 所示。假设该表中的示例基于如下前提条件:

```
>>> from random import *
>>> seq = ('a','e','i','o','u'); seq1 = [1, 2, 3, 4, 5]
```

表 14-4　random 模块中的随机序列函数

名　　称	说　　明	示　　例	结果(随机)
choice(seq)	从非空的序列 seq 中随机返回一个元素	for i in range(5): print(choice(seq),end=',')	e,e,e,a,e

续表

名　　称	说　　明	示　　例	结果（随机）
sample(population, k)	从非空的序列 population 中随机抽取 k 个元素，返回其列表	sample(seq,3)	['i', 'u', 'a']
shuffle(x[, random])	混排列表。可选的 random 为随机函数，默认为 random()	shuffle(seq1); seq1	[2, 1, 5, 3, 4]

【例 14.4】 混排示例（random_shuffle.py）：随机生成扑克牌的 4 手牌（4 个人的牌局，每手牌 13 张）。

```
import random
#一副牌:Club(梅花)、Diamond(方块)、Heart(红桃)、Spade(黑桃)、2~10,J,Q,K,A
cards = ['2C','3C','4C','5C','6C','7C','8C','9C','10C','JC','QC','KC','AC',
         '2D','3D','4D','5D','6D','7D','8D','9D','10D','JD','QD','KD','AD',
         '2H','3H','4H','5H','6H','7H','8H','9H','10H','JH','QH','KH','AH',
         '2S','3S','4S','5S','6S','7S','8S','9S','10S','JS','QS','KS','AS']
random.shuffle(cards)                                   #混排,洗牌
deck1 = [];deck2 = [];deck3 = [];deck4 = []             #初始化 4 手牌
for i in range(13):                                     #发牌
    deck1.append(cards.pop())
    deck2.append(cards.pop())
    deck3.append(cards.pop())
    deck4.append(cards.pop())
print(deck1);print(deck2);print(deck3);print(deck4)
```

程序运行结果如下（随机结果）。

```
['KC', '3H', '9H', 'AH', 'KH', '4H', 'QC', '6S', '2H', 'KD', '8C', '3D', '6H']
['9S', 'JC', '7H', 'AS', '5D', 'JS', '3S', 'KS', 'AC', 'JD', 'JH', '10H', '9D']
['7D', '4C', '7C', '8D', 'QD', '2S', '6D', '2C', '4S', 'QH', 'QS', '10S', '5H']
['4D', '5C', '3C', '6C', '8H', '7S', '9C', '5S', '8S', '10D', '2D', 'AD', '10C']
```

14.5 数值运算模块 NumPy

NumPy 库是 Python 的数值计算扩展，用于高效地存储和处理数组和矩阵，许多科学计算库（包括 Matplotlib、Pandas、SciPy 和 SymPy 等）都基于它。

NumPy 是 Python 数值计算的基石，它提供了两种基本的对象——ndarray（N-dimensional array object，N 维数组对象，即数组）和 ufunc（universal function object，通用函数对象）。ndarray 是存储单一数据类型的多维数组，而 ufunc 是能够对数组进行处理的通用函数。

14.5.1 数值运算模块的基本使用

Python 的列表可以用作数组，但由于列表的元素可以是任何对象，故列表中所保存的是对象的指针，对于数值运算会浪费内存和 CPU 计算时间。而 Python 标准库 array 不支持多维数值，也不支持数值运算函数，同样不适合数值计算。

Python 扩展模块 NumPy 具有数组和矩阵处理功能（类似于 MATLAB），提供了更高效的数值处理功能。

使用 NumPy 模块一般遵循如下几个步骤。

(1) 安装 NumPy 模块,具体步骤请参见本书第 1 章的相关内容。

(2) 使用 import numpy 语句导入 NumPy 模块。

(3) 创建数组。

(4) 处理数组。

14.5.2 创建数组

创建数组有如下几种方式。

(1) 通过 array()函数把序列对象参数转换为数组。

(2) 通过 arange()、linspace()和 logspace()函数创建数组。

【例 14.5】 通过 array()函数创建数组示例。

```
>>> import numpy as np
>>> a = np.array([1,2,3])                              #一维数组
>>> b = np.array([[1,2,3],[4,5,6],[7,8,9]])            #二维数组
>>> a                                                  #输出:array([1, 2, 3])
>>> b
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

说明:如果传递给 array 对象的参数是多层嵌套的序列,将创建多维数组。

【例 14.6】 通过 arange()、linspace()和 logspace()函数创建数组示例。

```
>>> a = np.arange(0,10,2)
>>> a                                                  #输出:array([0, 2, 4, 6, 8])
>>> b = np.linspace(0, 2 * np.pi, 10)
>>> b
array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
       3.4906585 ,  4.1887902 , 4.88692191, 5.58505361, 6.28318531])
>>> c = np.logspace(0, 2, 10)
>>> c
array([  1.        ,   1.66810054,   2.7825594 ,   4.64158883,
         7.74263683,  12.91549665,  21.5443469 ,  35.93813664,
         59.94842503, 100.])
```

说明:

(1) arange()函数通过指定开始值、终值和步长来创建一维数组。

(2) linspace()函数通过指定开始值、终值和元素个数来创建一维数组,可以通过 endpoint 命名参数指定是否包括终值,默认设置是包括终值。

(3) logspace()函数和 linspace()函数类似,用于创建等比数列。

14.5.3 处理数组

数组元素的存取方法和 Python 列表的存取方法相同,但是通过下标范围获取的是原始数组的一个视图,与原始数组共享同一块数据空间,这与 Python 的列表切片操作不同。

数组也支持常用的运算符操作,例如+、-、*、/等。

NumPy 内置了许多针对数组的每个元素分别进行操作的函数,这些函数称为 universal function,它们一般基于 C 语言级别实现,因此其计算速度非常快。

【例 14.7】 数组处理示例。

```
>>> import numpy as np
>>> x = np.linspace(0, 2 * np.pi, 10)
>>> y = np.sin(x)
>>> y
array([ 0.00000000e+00,    6.42787610e-01,   9.84807753e-01,   8.66025404e -01,
         3.42020143e-01,  -3.42020143e -01,  -8.66025404e-01,  -9.84807753e-01,
       -6.42787610e-01,  -2.44929360e-16])
>>> y+1
array([1.          , 1.64278761, 1.98480775, 1.8660254, 1.34202014,
       0.65797986,  0.1339746 , 0.01519225, 0.35721239, 1.          ])
```

14.5.4 数组应用举例

使用模块 NumPy 中的函数可以实现数值处理,结合 Matplotlib 中的绘图函数可以很方便地输出各种处理结果。

【例 14.8】 数组应用示例(funcfig.py):利用 NumPy 模块中的函数和 Matplotlib 中的绘图函数绘制正弦和余弦函数图形。

```
import numpy as np
import matplotlib.pyplot as plt
import math
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, x, y2)
plt.show()
```

程序运行结果如图 14-2 所示。

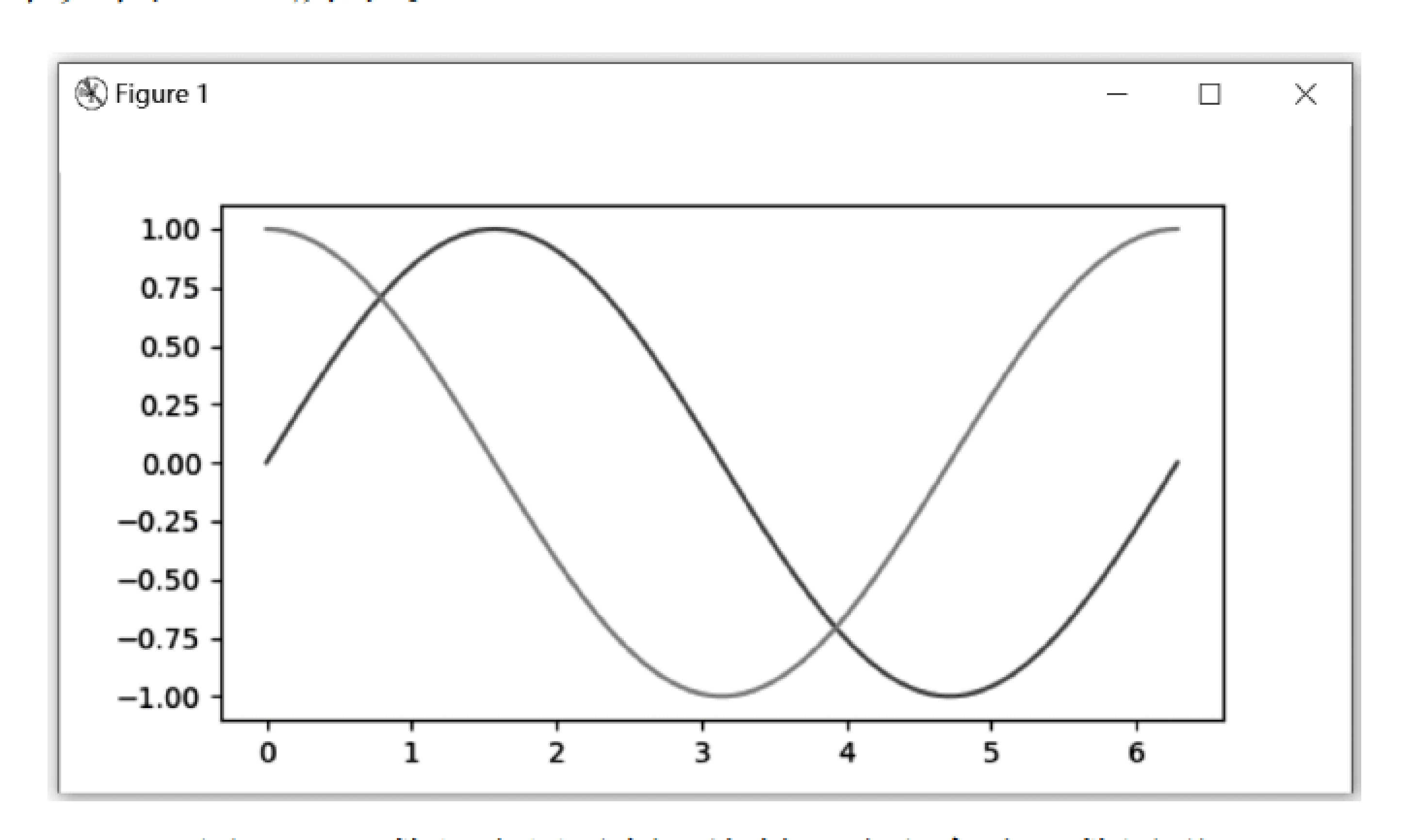


图 14-2 数组应用示例:绘制正弦和余弦函数图形

14.6 日期和时间处理

14.6.1 相关术语

1. epoch

epoch(新纪元)是系统规定的时间起始点。UNIX 系统于 1970/1/1 0:0:0 开始。日期和时间在内部表示为从 epoch 开始的秒数。time 模块中的函数使用对应 C 语言函数库中的

函数，故只能处理 1970/1/1 和 2038/12/31 之间的日期和时间。

使用 time 模块的 getime()函数可以获取当前系统的起始点。例如：

```
>>> import time; time.getime(0)
time.struct_time(tm_year = 1970, tm_mon = 1, tm_mday = 1, tm_hour = 0, tm_min = 0, tm_sec = 0, tm_wday = 3, tm_yday = 1, tm_isdst = 0)
```

2. UTC

UTC 是 Coordinated Universal Time(协调世界时)的英文缩写，是一种兼顾理论与应用的时标，旧称 GMT(Greenwich Mean Time，格林尼治时间)。

3. DST

DST(Daylight Saving Time)即夏令时。不同地域可能规定不同的夏令时，C 语言函数库使用表格对应这些规定。time 模块中的 daylight 属性用于判定是否使用夏令时。

```
>>> time.daylight                                    #输出:0
```

14.6.2 时间对象

time 模块的 struct_time 对象是一个命名元组，用于表示时间对象，包括 9 个字段属性，即 tm_year、tm_mon、tm_mday、tm_hour、tm_min、tm_sec、tm_wday、tm_yday、tm_isdst。

time 模块的 getime()、localtime()和 strptime()函数返回 struct_time 对象。

例如：

```
>>> st = time.getime()
>>> st.tm_year                                       #输出:2018
```

14.6.3 测量程序运行时间

time 模块包含以下用于测量程序性能的函数。

- process_time()：返回当前进程处理器运行时间。
- perf_counter()：返回性能计数器。
- monotonic()：返回单向时钟。

可以使用程序运行到某两处的时间差值计算该程序片段所花费的运行时间，也可以使用 time.time()函数，该函数返回以秒为单位的系统时间(浮点数)。

【例 14.9】 测量程序运行时间(time_pmrunning.py)。

```
import time
def test():
    sum = 0
    for i in range(0,9999999):
        sum += i
    return sum
if __name__ == '__main__':
    t1 = time.monotonic()                            #单向时钟
    print(test())
    t2 = time.monotonic()                            #单向时钟
    print('运行时间:', t2 - t1)
```

程序运行结果如下。

```
49999985000001
运行时间: 0.594000000118744
```

14.6.4 日期对象

datetime 模块包括 datetime.MINYEAR 和 datetime.MAXYEAR 两个常量，分别表示最小年份和最大年份，值为 1 和 9999。

datetime 模块中包含用于表示日期的 date 对象、表示时间的 time 对象和表示日期时间的 datetime 对象。timedelta 对象表示日期或时间之间的差值，可以用于日期或时间的运算。

【例 14.10】 日期对象示例。

```
>>> import datetime
>>> datetime.date.today()                    #当前日期.输出:datetime.date(2018, 7, 9)
>>> datetime.datetime.now()                  #当前日期时间
datetime.datetime(2018, 7, 9, 21, 17, 3, 983307)
```

14.6.5 获取当前日期时间

通过 datetime 模块中的 date.today()函数可以返回表示当前日期的 date 对象，通过其实例对象方法可以获取其年、月、日等信息。

通过 datetime 模块中的 datetime.now()函数可以返回表示当前日期时间的 datetime 对象，通过其实例对象方法可以获取其年、月、日、时、分、秒等信息。

【例 14.11】 获取当前日期时间示例(datetimes.py)。

```
import datetime
d = datetime.date.today()
dt = datetime.datetime.now()
print ("当前的日期是 %s" % d)
print ("当前的日期和时间是 %s" % dt)
print ("ISO 格式的日期和时间是 %s" % dt.isoformat() )
print ("当前的年份是 %s" %dt.year)
print ("当前的月份是 %s" %dt.month)
print ("当前的日期是  %s" %dt.day)
print ("dd/mm/yyyy 格式是  %s/%s/%s" % (dt.day, dt.month, dt.year) )
print ("当前小时是 %s" %dt.hour)
print ("当前分钟是 %s" %dt.minute)
print ("当前秒是  %s" %dt.second)
```

程序运行结果如图 14-3 所示。

```
当前的日期是 2018-07-09
当前的日期和时间是 2018-07-09 21:17:42.165847
ISO格式的日期和时间是 2018-07-09T21:17:42.165847
当前的年份是 2018
当前的月份是 7
当前的日期是  9
dd/mm/yyyy 格式是  9/7/2018
当前小时是 21
当前分钟是 17
当前秒是  42
```

图 14-3 获取当前日期时间示例的运行结果

14.6.6 日期时间格式化为字符串

time 模块中的 strftime()函数用于将 struct_time 对象格式化为字符串，其函数形式如下：

```
time.strftime(format[, t])
```

其中,format 为日期格式化字符串;可选参数 t 为 struct_time 对象。

【例 14.12】　日期时间格式化为字符串示例 1。

```
>>> from time import *
>>> strftime("%c", localtime())                    #输出:'Mon Jul  9 21:18:59 2018'
>>> strftime("%Y年%m月%d日(%A) %H时%M分%S秒", localtime())
'2018年07月09日(Monday) 21时19分19秒'
```

datetime.datetime.strftime()函数用于将 datetime 对象格式化为日期时间字符串。

【例 14.13】　日期时间格式化为字符串示例 2。

```
>>> import datetime
>>> datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
'2018-07-09 21:19:43'
```

14.6.7　日期时间字符串解析为日期时间对象

time 模块中的 strptime()函数用于将时间字符串解析为 struct_time 对象,其函数形式如下:

time.strptime(string[, format])

其中,string 为日期字符串;可选参数 format 为日期格式化字符串。

【例 14.14】　日期时间字符串解析示例 1。

```
>>> from time import *
>>> strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0, tm_sec=0, tm
_wday=3, tm_yday=335, tm_isdst=-1)
```

datetime.datetime.strptime()用于将日期时间字符串解析为 datetime 对象,其函数形式如下:

datetime.datetime.strptime(string, format)

其中,string 为日期字符串;format 为日期格式化字符串。

【例 14.15】　日期时间字符串解析示例 2。

```
>>> datetime.datetime.strptime('2019-08-18','%Y-%m-%d')
datetime.datetime(2019, 8, 18, 0, 0)
```

14.7　应 用 举 例

14.7.1　蒙特卡洛模拟:赌徒破产命运

赌徒的最终命运是破产:假设一个赌徒从给定赌资筹码开始,连续下注一系列的 1 个筹码的赌注,最终结果赌徒注定会输光。

可以通过随机模拟来证明该假设,假设初始赌资为 stake 个筹码,每次下注通过随机数 0 和 1 来判断输赢:0 的时候筹码加 1,1 的时候筹码减 1,当筹码小于 0 时终止。模拟 trials 取平均下注次数。

【例 14.16】　赌徒破产命运(gambler1.py)。

```
import random
def gamble(stake, trials):
```

```
    """返回输掉 stake 所需要的次数,模拟 trials 次取平均值"""
    total_bets = 0                                    #总下注次数
    max_cash = stake                                  #最大赌资
    for t in range(trials):                           #模拟 trials 次取平均
        cash = stake                                  #筹码
        while cash > 0:                               #持续下注直到破产
            #模拟一次下注
            total_bets += 1
            if random.randrange(0, 2) == 0:
                cash += 1
                max_cash = max(max_cash, cash)
            else: cash -= 1
            #print(cash)
    #print("赌博过程中最大赌资 = {}".format(max_cash))
    return int(total_bets/trials)
if __name__ == "__main__":
    print("输掉{}个筹码的平均次数:{}".format(1,gamble(1,100)))
    print("输掉{}个筹码的平均次数:{}".format(5,gamble(5,100)))
    print("输掉{}个筹码的平均次数:{}".format(10,gamble(10,100)))
    print("输掉{}个筹码的平均次数:{}".format(20,gamble(20,100)))
```

程序运行结果如下。

```
输掉 1 个筹码的平均次数:26
输掉 5 个筹码的平均次数:2766
输掉 10 个筹码的平均次数:14353
输掉 20 个筹码的平均次数:32591
```

运行程序每次结果不一样，但最终都会完成并输出破产需要的次数。注意，如果 stake 数比较大，则程序运行的时间可能比较长。

修改上述程序，可以计算给定初始赌资筹码，最终赢得目标筹码的概率（赢得一定数额后离场）。

【例 14.17】 赌徒赢的概率(gambler2.py)。

```
import random
def gamble(stake, goal, trials):
    """返回输掉 stake 所需要的次数,模拟 trials 次取平均值"""
    bets = 0                                 #总下注次数
    wins = 0                                 #赢的次数
    for t in range(trials):                  #模拟 trials 次取平均
        cash = stake                         #筹码
        #持续下注直到破产,或达到目标退场
        while cash > 0 and cash < goal:
            #模拟一次下注
            bets += 1
            if random.randrange(0, 2) == 0:
                cash += 1
            else: cash -= 1
        if cash >= goal: wins += 1           #赢的次数
    return wins/trials, int(bets/trials)
if __name__ == "__main__":
    p, n = gamble(10,20,100)
    print("{}赢{}的概率{}%,平均下注次数{}".format(10,20,p*100,n))
    p, n = gamble(10,1000,100)
    print("{}赢{}的概率{}%,平均下注次数{}".format(10,1000,p*100,n))
```

程序运行结果如下(注：每次运行结果不完全相同)。

```
10 赢 20 的概率 40.0%,平均下注次数 93
10 赢 1000 的概率 1.0%,平均下注次数 14583
```

14.7.2　使用随机数估值圆周率

使用随机数估值圆周率的一种数学方法如下：随机生成 n 个坐标点(x,y),x 和 y 位于－1 到 1 之间,对应于一个 2×2 的正方形,如图 14-4 所示。假设其中有 k 个点位于正方形的内切圆之内,则当 n 足够大时,k 与 n 之比近似于圆的面积和正方形的面积之比,即 k/n = (π＊1＊1)/(2＊2),因此 π=4＊k/n。

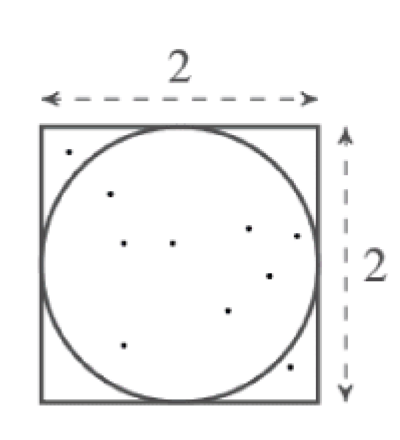


图 14-4　随机数估值圆周率

【例 14.18】　使用随机数估值圆周率(pi.py)。

```
import random
def pi(trials):
    """通过随机点位置近似计算圆周率 pi"""
    hits = 0                              #命中圆环的点的数量
    for i in range(trials):
        x = random.uniform(-1,1)          #随机生成-1 到 1 之间的 x 坐标
        y = random.uniform(-1,1)          #随机生成-1 到 1 之间的 y 坐标
        if x**2+y**2<=1:                  #如果位于圆环内,则命中数加 1
            hits += 1
    return 4*hits/trials
#测试代码
if __name__ == "__main__":
    for i in range(5):
        n = 10000*(10**i)
        print("试验{}次后计算近似圆周率为:{}".format(n, pi(n)))
```

程序运行结果如下。

```
试验 10000 次后计算近似圆周率为:3.1028
试验 100000 次后计算近似圆周率为:3.14904
试验 1000000 次后计算近似圆周率为:3.142096
试验 10000000 次后计算近似圆周率为:3.1414452
试验 100000000 次后计算近似圆周率为:3.1417788
```

14.7.3　程序运行时间测量

从 14.7.1 节和 14.7.2 节例子的运行过程可以发现,当程序的复杂度增加(例如循环次数增大)时,程序运行的时间显著增加。通过 time 模块中的 time()函数可以测量程序运行的时间。

【例 14.19】　程序运行时间测量(timing.py)。

```
import time
def timing(f, data):
    """测量函数调用 f(data)的运行时间分析"""
    start = time.time()                   #记录开始时间
    f(data)                               #运行 f(data)
    end = time.time()                     #记录结束时间
    return end - start                    #返回执行时间
#测试代码
if __name__ == "__main__":
```

```
import pi                                    #参见 14.7.2 节
for i in range(3):
    n = 10000 * (100 * * i)
    t = timing(pi.pi, n)
    print("pi({})的运行时间为:{}".format(n, t))
```

程序运行结果如下。

```
pi(10000)的运行时间为:0.01856255531311035
pi(1000000)的运行时间为:1.1727638244628906
pi(100000000)的运行时间为:120.24882817268372
```

14.8 复 习 题

一、填空题

1. Python 中的________模块包含各种用于日期和时间处理的类;________模块包含用于处理日历的函数和类;________模块包含用于处理时间的函数。

2. datetime 模块中包含用于表示日期的________对象、表示时间的________对象、表示日期时间的________对象、表示日期或时间之间差值的________对象、表示时区信息的________对象和________对象。

3. 使用 time 模块中的________函数可以获取当前系统的起始点。

4. time 模块中的________属性用于判定是否使用夏令时。

5. time 模块中的________函数用于将字符串解析为 struct_time 对象;________函数用于将 struct_time 对象格式化为字符串。

6. datetime 模块包括________和________两个常量,分别表示最小年份和最大年份,值为________和________。

7. date、time 和 datetime 对象的________方法将 struct_time 对象格式化为字符串;datetime 类方法________将字符串解析为 datetime 对象。

8. timedelta 对象 td 的属性________获取天数,________获取秒数,________获取毫秒数。

9. Python 中 calendar.isleap(2000)的结果为________。

二、思考题

1. 下列 Python 语句的执行结果是________。

```
from datetime import * ; import time,datetime
print(date.min,date.max,date.fromordinal(32))
d = date(2019, 10, 1);print(d.year, d.month, d.day);d.replace(month=12)
print(d.toordinal(),d.weekday(),d.ctime(),d.strftime("%Y/%m/%d(%a)"))
```

2. 下列 Python 语句的执行结果是________。

```
import datetime; t = datetime.time(19, 30, 45,196)
print(datetime.time.min,datetime.time.max)
print(t.hour, t.minute, t.second,t.microsecond)
t.replace(hour=23);print(t.strftime("%H时%M分%S秒"))
```

3. 下列 Python 语句的执行结果是________。

```
import datetime; dt = datetime.datetime(2019, 5, 1, 9, 35, 46)
print(datetime.datetime.min,datetime.datetime.max)
```

```
print(dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second)
print(dt.date(),dt.time(),dt.strftime("%Y/%m/%d(%A),%H时%M分%S秒"))
```

4. 下列 Python 语句的执行结果是________。

```
from datetime import * ; td1 = timedelta(minutes=10)
td2 = timedelta(minutes=15); print(td1+td2,(td2-td1).seconds,td1*10,td1<td2)
```

5. 下列 Python 语句的执行结果是________。

```
from datetime import * ; dt1 = date(2019, 6, 1); dt2 = date(2019,5,1)
td = timedelta(days=10); print((dt1-dt2).days,dt1+td,dt1-td,dt1>dt2)
```

14.9　上 机 实 践

1. 完成本章中的例 14.1～例 14.19,熟悉 Python 语言中的数值处理、日期和时间处理程序设计。

2. 编写程序,打印 2018 年 1—12 月份的日历,运行效果如图 14-5 所示。

```
                                  2018

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                1  2  3  4
 8  9 10 11 12 13 14       5  6  7  8  9 10 11       5  6  7  8  9 10 11
15 16 17 18 19 20 21      12 13 14 15 16 17 18      12 13 14 15 16 17 18
22 23 24 25 26 27 28      19 20 21 22 23 24 25      19 20 21 22 23 24 25
29 30 31                  26 27 28                  26 27 28 29 30 31

       April                      May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1          1  2  3  4  5  6                   1  2  3
 2  3  4  5  6  7  8       7  8  9 10 11 12 13       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      14 15 16 17 18 19 20      11 12 13 14 15 16 17
16 17 18 19 20 21 22      21 22 23 24 25 26 27      18 19 20 21 22 23 24
23 24 25 26 27 28 29      28 29 30 31               25 26 27 28 29 30
30

        July                     August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                      1  2
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       3  4  5  6  7  8  9
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      10 11 12 13 14 15 16
16 17 18 19 20 21 22      20 21 22 23 24 25 26      17 18 19 20 21 22 23
23 24 25 26 27 28 29      27 28 29 30 31            24 25 26 27 28 29 30
30 31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                      1  2
 8  9 10 11 12 13 14       5  6  7  8  9 10 11       3  4  5  6  7  8  9
15 16 17 18 19 20 21      12 13 14 15 16 17 18      10 11 12 13 14 15 16
22 23 24 25 26 27 28      19 20 21 22 23 24 25      17 18 19 20 21 22 23
29 30 31                  26 27 28 29 30            24 25 26 27 28 29 30
                                                    31
```

图 14-5　2018 年日历运行效果

提示：

参考代码如图 14-6 所示。

```
import calendar
import locale
textcal = calendar.TextCalendar() #创建文本日历
textcal.pryear(2018)              #打印2018年一年的日历
loc = locale.getlocale()                            #获取当前系统的locale（本地化配置）
localtextcal = calendar.LocaleTextCalendar(locale=loc) #返回指定locale的月份和星期信息
```

图 14-6　2018 年日历参考代码

3. 编写程序,定义一个返回指定年月的天数的函数 ndays(y,m),并编写测试代码,运行效果如图 14-7 所示。

提示：

（1）可以使用 calendar 模块的 isleap()函数来判断闰年。

（2）参考代码如图 14-8 所示。

```
请输入年份（>=1），否则为1：2018
请输入月份（1~12），否则<1为1、>12为12：12
31
```

图 14-7　返回指定年月的天数的程序运行效果

```
from calendar import *
def ndays(y,m):
    #每个月的正常天数
    monthDay=[31,28,31,30,31,30,31,31,30,31,30,31]
    days = monthDay[ m-1]
    if (m==2 and isleap(y)):
        days+=1
    return(days)
#测试代码
y=int(input("请输入年份（>=1），否则为1："))
m=int(input("请输入月份（1~12），否则<1为1、>12为12："))
if y<1: y=1
if m<1: m=1
if m>12: m=12
print(ndays(y,m))
```

图 14-8　返回指定年月的天数的程序参考代码

4. 编写程序，定义一个返回从公元 1 年 1 月 1 日（含）到 y 年 m 月 d 日（含）的天数的函数 caldays(y,m,d)，并编写测试代码。其运行效果如图 14-9 所示。

提示：

计算从公元 1 年 1 月 1 日到 y 年 m 月 d 日的天数，可以分为 3 个部分计算。

（1）计算从公元 1 年到 y－1 年的天数，每年是 365 天或 366 天（闰年）。

（2）对于第 y 年，先计算 1～m－1 月整月的天数，可利用上一题返回指定年月的天数的函数 ndays(y,m)。

（3）最后加上零头（第 m 月的 d 天）。

5. 编写程序，定义一个返回指定年月日的星期的函数 weekday(y,m,d)，结果为星期一、……、星期日，并编写测试代码。其运行效果如图 14-10 所示。

```
请输入年份（>=1），否则为1：2018
请输入月份（1~12），否则<1为1、>12为12：12
请输入日期（1~31），否则<1为1、>ndays(y,m)为ndays(y,m)：5
从1年1月1日到 2018 年 12 月 5 日共 737033 天
```

图 14-9　返回总天数的运行效果

```
请输入年份（>=1），否则为1：2018
请输入月份（1~12），否则<1为1、>12为12：12
请输入日期（1~31），否则<1为1、>ndays(y,m)为ndays(y,m)：5
2018 年 12 月 5 日是 星期三
```

图 14-10　返回星期的运行效果

提示：

如果要求出指定年月日是星期几，只需调用上一题返回从公元 1 年 1 月 1 日到 y 年 m 月 d 日的天数的函数 caldays(y,m,d)，再除以 7，得到的余数即为星期几，余 0 就是星期日。

14.10　案例研究：使用 pandas 进行数据分析和处理

pandas 是基于 NumPy 的 Python 数据分析库（Python Data Analysis Library）。pandas 提供了大量处理数据的函数和方法，可以高效地进行数据分析和处理。

本章案例研究通过 pandas 的几个简单应用例子引导读者进入数据分析和处理的大门。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 15 章

字符串和文本处理



视频讲解

Python 提供了丰富的数据类型和模块函数，用于程序设计中的字符串和文本处理。

15.1 相关模块概述

15.1.1 字符串和文本处理的相关模块

1. Python 标准库中的字符串和文本处理相关模块

- string 模块：包含若干字符集常量，其处理字符串的函数已经被字符串对象的方法替代。
- re 模块：正则表达式处理。
- codecs 模块：字符编码处理。
- difflib 模块：比较字符串列表的差异。
- gettext 模块：语言国际化。
- textwrap 模块：格式化文本段落。
- unicodedata 模块：Unicode 字符库。

2. 自然语言处理模块库(NLTK)

NLTK 是 Python 用于自然语言处理(Natural Language Processing，NLP)的第三方库，使用 NLTK 可以完成很多自然语言处理任务，包括分词、词性标注、命名实体识别及句法分析等。

15.1.2 字符串处理的常用方法

在程序设计过程中往往涉及文本的分析和处理，Python 提供了如下字符串处理常用方法。

(1) 使用 str 对象提供的方法：可以实现常用的字符串处理功能。

(2) 使用正则表达式：匹配和查找字符串并对其进行相应的修改处理。

(3) 使用专用的第三方文本处理模块(例如 NLTK)。

15.2 字符串处理的常用操作

15.2.1 字符串的类型判断

str 对象包括如下用于判断字符串类型的方法。

- str.isalnum()：是否全为字母或数字。
- str.isalpha()：是否全为字母。

- str.isdecimal():是否只包含十进制数字字符。
- str.isdigit():是否全为数字(0~9)。
- str.isidentifier():是否为合法标识。
- str.islower():是否全为小写。
- str.isupper():是否全为大写。
- str.isnumeric():是否只包含数字字符。
- str.isprintable():是否只包含可打印字符。
- str.isspace():是否只包含空白字符。
- str.istitle():是否为标题,即各单词首字母大写。

【例 15.1】 字符串类型判断示例。

>>> s1='yellow ribbon'
>>> s2='Pascal Case'
>>> s3='123'
>>> s4='iPhone7'
	>>> s1.islower()
	True
	>>> s2.isupper()
	False
		>>> s4.isalnum()
		True
		>>> s3.isnumeric()
		True
			>>> s1.isdigit()
			False
			>>> s2.istitle()
			True

15.2.2 字符串的大小写转换

str 对象包括如下用于字符串大小写转换的方法。

- str.capitalize():转换为首字母大写,其余小写。
- str.lower():转换为小写。
- str.upper():转换为大写。
- str.swapcase():大小写互换。
- str.title():转换为各单词首字母大写。
- str.casefold():转换为大小写无关字符串比较的格式字符串。

【例 15.2】 字符串大小写转换示例。

>>> s1='red car'
>>> s2='Pascal Case'
>>> s3='python3.7'
>>> s4='iPhoneX'
	>>> s1.capitalize()
	'Red car'
	>>> s2.lower()
	'pascal case'
		>>> s3.upper()
		'PYTHON3.7'
		>>> s2.swapcase()
		'pASCAL cASE'
			>>> s1.title()
			'Red Car'
			>>> s4.casefold()
			'iphonex'

15.2.3 字符串的填充、空白和对齐

str 对象包括如下用于填充、空白和对齐字符串的方法。

- str.strip([chars]):去两边空格,也可指定要去除的字符列表。
- str.lstrip([chars]):去左边空格,也可指定要去除的字符列表。
- str.rstrip([chars]):去右边空格,也可指定要去除的字符列表。
- str.zfill(width):左填充,使用 0 填充到 width 长度。
- str.center(width[, fillchar]):两边填充,使用填充字符 fillchar(默认空格)填充到 width 长度。
- str.ljust(width[, fillchar]) :左填充,使用填充字符 fillchar(默认空格)填充到 width

长度。

- str.rjust(width[, fillchar])：右填充，使用填充字符 fillchar（默认空格）填充到 width 长度。
- str.expandtabs([tabsize])：将字符串中的制表符（tab）扩展为若干个空格，tabsize 默认为 8。

【例 15.3】 字符串填充、空白和对齐示例。

>>> s1='123'
>>> s2=' 123 '
>>> len(s2)
6
	>>> s2.strip()
	'123'
	>>> s2.lstrip()
	'123 '
		>>> s1.zfill(5)
		'00123'
		>>> s1.center(5, ' ')
		' 123 '
			>>> s1.ljust(5)
			'123 '
			>>> s1.rjust(5, '0')
			'00123'

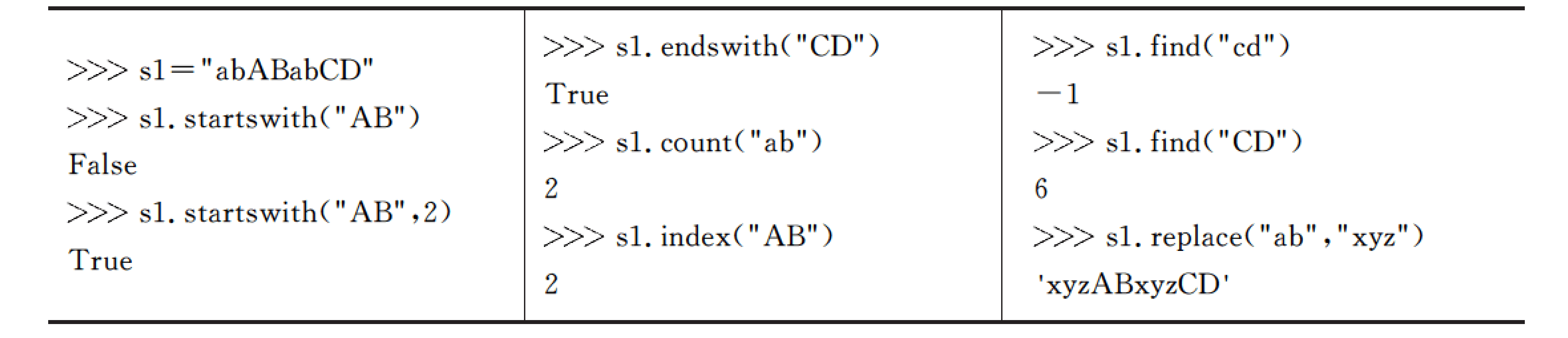
15.2.4 字符串的测试、查找和替换

str 对象包括如下用于字符串测试、查找和替换的方法。

- str.startswith(prefix[, start[, end]])：是否以 prefix 开头。
- str.endswith(suffix[, start[, end]])：是否以 suffix 结尾。
- str.count(sub[, start[, end]])：返回指定字符串出现的次数。
- str.index(sub[, start[, end]])：搜索指定字符串，返回下标，没有则导致 ValueError。
- str.rindex(sub[, start[, end]])：从右边开始搜索指定字符串，返回下标，没有则导致 ValueError。
- str.find(sub[, start[, end]])：搜索指定字符串，返回下标，没有则返回－1。
- str.rfind(sub[, start[, end]])：从右边开始搜索指定字符串，返回下标，没有则返回－1。
- str.replace(old, new[, count])：替换 old 为 new，可选 count 为替换次数。

其中，可选指定范围[start, end)为从下标 start（包括 start，默认为 0）开始到下标 end 结束（不包括 end，默认为 len(s)）。

【例 15.4】 字符串测试、查找和替换示例。



>>> s1="abABabCD"
>>> s1.startswith("AB")
False
>>> s1.startswith("AB",2)
True
	>>> s1.endswith("CD")
	True
	>>> s1.count("ab")
	2
	>>> s1.index("AB")
	2
		>>> s1.find("cd")
		－1
		>>> s1.find("CD")
		6
		>>> s1.replace("ab","xyz")
		'xyzABxyzCD'

15.2.5 字符串的拆分和组合

str 对象包括如下用于字符串拆分和组合的方法。

- str.split(sep=None, maxsplit=－1)：按指定字符（默认为空格）分隔字符串，返回列表 maxsplit 为最大分隔次数，默认为－1，无限制。
- str.rsplit(sep=None, maxsplit=－1)：从右侧按指定字符分隔字符串，返回列表。

- str.partition(sep)：根据分隔符 sep 分隔字符串为两部分，返回元组（left，sep，right）。
- str.rpartition(sep)：根据分隔符 sep 从右侧分隔字符串为两部分，返回元组（left，sep，right）。
- str.splitlines([keepends])：按行分隔字符串，返回列表。
- str.join(iterable)：将 iterable 中的各元素组合成字符串，若包含非字符串元素，则导致 TypeError。

【例 15.5】 字符串拆分和组合示例。

>>> s1='one,two,three'	>>> s1.partition(',')	>>> s2.splitlines()	>>> s4=':'
>>> s1.split(',')	('one', ',', 'two,three')	['abc', '123', 'xyz']	>>> s4.join(s3)
['one', 'two', 'three']	>>> s1.rpartition(',')	>>> s2.splitlines(True)	'a:b:c'
>>> s1.rsplit(',', 1)	('one,two', ',', 'three')	['abc\n', '123\n', 'xyz']	>>> s4.join('123')
['one,two', 'three']	>>> s2='abc\n123\nxyz'	>>> s3=('a','b','c')	'1:2:3'

15.2.6 字符串的翻译和转换

str 对象包括如下用于字符串翻译和转换的方法。

- static str.maketrans(x[, y[, z]])：创建用于 translate 的转换表。
- str.translate(map)：根据 map 转换。

【例 15.6】 字符串翻译和转换示例。

>>> table1=str.maketrans('1234567','一二三四五六日')	>>> weeks={'1':'M一','2':'T二','3':'W三','4':'T四','5':'F五','6':'S六','7':'S日'}
>>> s1='1 3 4 9'	>>> table2=str.maketrans(weeks)
>>> s1.translate(table1)	>>> s1.translate(table2)
'一 三 四 9'	'M一 W三 T四 9'

15.2.7 字符串应用举例

【例 15.7】 字符串的使用示例 1(str_count.py)：输入任意字符串，统计其中元音字母（'a'、'e'、'i'、'o'、'u'，不区分大小写）出现的次数和频率。

```
s1 = input('请输入字符串:')                 # 'The quick brown fox jumps over the lazy dog'
s2 = s1.upper()                             # 转换为大写
countall = len(s1)                          # 字符串长度
counta = s2.count('A');counte = s2.count('E');counti = s2.count('I')
counto = s2.count('O');countu = s2.count('U')
print('所有字母的总数为:', countall)
print('元音字母出现的次数和频率分别为:')
print('A:{0}\t{1:2.2f}%'.format(counta, counta/countall * 100))
print('E:{0}\t{1:2.2f}%'.format(counte, counte/countall * 100))
print('I:{0}\t{1:2.2f}%'.format(counti, counti/countall * 100))
print('O:{0}\t{1:2.2f}%'.format(counto, counto/countall * 100))
print('U:{0}\t{1:2.2f}%'.format(countu, countu/countall * 100))
```

程序运行结果如下。

```
请输入字符串:The quick brown fox jumps over the lazy dog
所有字母的总数为: 43
元音字母出现的次数和频率分别为:
A:1  2.33%
E:3  6.98%
I:1  2.33%
O:4  9.30%
U:2  4.65%
```

【例 15.8】 字符串的使用示例 2(txt_count.py)：读取文本文件，统计其中的行数、字符数和单词个数。

```
file_name = "txt_count.py"                  #文本文件名
line_counts = 0                             #行数
word_counts = 0                             #单词个数
character_counts = 0                        #字符数
with open(file_name, 'r',encoding='utf8') as f:
    for line in f:
        words = line.split()                #分离出单词
        line_counts += 1                    #行数加 1
        word_counts += len(words)           #单词个数加 1
        character_counts += len(line)       #字符数加 1
print("行数:", line_counts)
print("单词个数:", word_counts)
print("字符个数:", character_counts)
```

程序运行结果如下。

```
行数: 13
单词个数: 47
字符个数: 470
```

15.3　正则表达式

正则表达式提供了功能强大、灵活又高效的方法来处理文本：快速分析大量文本以找到特定的字符模式；提取、编辑、替换或删除文本子字符串；将提取的字符串添加到集合以生成报告。正则表达式广泛用于各种字符串处理应用程序，例如 HTML 处理、日志文件分析和 HTTP 标头分析等。

15.3.1　正则表达式语言概述

在进行文本字符串处理时经常需要查找符合某些复杂规则(也称之为模式)的字符串，正则表达式语言就是用于描述这些规则(模式)的语言，使用正则表达式可以匹配和查找字符串，并对其进行相应的修改处理。

正则表达式是由普通字符(例如字符 a 到 z)以及特殊字符(称为元字符)组成的文字模式，元字符包括.、^、$、*、+、?、{、}、[、]、\、|、(、)。例如：

```
"Go"        #匹配字符串"God Good"中的"Go"
"G.d"       #匹配字符串"God Good"中的"God",.为元字符,匹配除行终止符以外的任何字符
"d$"        #匹配字符串"God Good"中的最后一个"d",$为元字符,匹配结尾
```

正则表达式的模式可以包含普通字符(包括转义字符)、字符类和预定义字符类、边界匹配符、重复限定符、选择分支、分组和引用等。

15.3.2 正则表达式引擎

正则表达式引擎是一种可以处理正则表达式的软件,流行的计算机语言都包含支持正则表达式处理的类库。Python 的模块 re 实现了正则表达式处理的功能,在导入 re 模块后,使用如下 findall()、search()函数可以进行匹配。

- re.findall(pattern, string):返回匹配结果列表。
- re.search(pattern, string):如果匹配,返回 Match 对象,否则返回 None。

【例 15.9】 正则表达式示例。

```
>>> import re                          #导入模块 re
>>> re.findall('d', 'godness')         #输出:['d']
```

15.3.3 普通字符和转义字符

最基本的正则表达式由单个或多个普通字符组成,用于匹配字符串中对应的单个或多个普通字符。普通字符包括 ASCII 字符、Unicode 字符和转义字符。

另外,正则表达式中的元字符(.、^、$、*、+、?、{、}、[、]、\、|、(、))包含特殊含义,如果要作为普通字符使用,则需要转义,例如\$。

```
>>> re.findall("fo", "the quick brown fox jumps for food") #输出:['fo', 'fo', 'fo']
>>> re.findall("1+1=2", "1+1=2")          #元字符+重复一次或多次,即匹配 11=
                                          #2,输出:[]
>>> re.findall("1+1=2", "11=2")           #元字符+重复一次或多次,即匹配 11=
                                          #2,输出:['11=2']
>>> re.findall("1\+1=2", "1+1=2")         #转义元字符+,即匹配 1+1=2,输出:
                                          #['1+1=2']
>>> re.findall("(note)", "please (note)")     #()在正则表达式中为分组,输出:['note']
>>> re.findall("\(note\)", "please (note)")   #\(匹配圆括号,输出:['(note)']
```

注意: 在正则表达式中包含特殊字符,例如\b 表示单词边界;而字符串中的转义字符\b 表示退格字符。因此在正则表达式中,这些与标准转义字符重复的特殊符号必须使用两个反斜线字符('\\'),或者使用原始字符串 r""或 r''。例如:

```
>>> re.findall("\bon\b", "only on air")       #\b 匹配退格符,输出:[]
>>> re.findall("\\bon\\b", "only on air")     #\\b 匹配单词边界,输出:['on']
>>> re.findall(r"\bon\b", "only on air")      #使用原始字符串,\b 匹配单词边界,输
                                              #出:['on']
```

15.3.4 字符类和预定义字符类

1. 字符类

字符类是由一对方括号[]括起来的字符集合,正则表达式引擎匹配字符集中的任意一个字符。字符类包括如下定义方式。

- [xyz]:枚举字符集,匹配括号中的任意字符。例如,"t[aeio]n"匹配"tan"、"ten"、"tin"、"ton"。

- [^xyz]：否定枚举字符集，匹配不在此括号中的任意字符。例如[^aeiou]。
- [a-z]：指定范围的字符，匹配指定范围的任意字符。例如[0-9]。
- [^a-z]：指定范围以外的字符，匹配指定范围以外的任意字符。例如[^0-9]

例如：

```
>>> re.findall("fo[xr]", "the quick brown fox jumps for food")      #输出:['fox', 'for']
```

2. 预定义字符类

在使用正则表达式时经常用到一些特定的字符类，例如数字字母。正则表达式语言包含若干预定义字符类，这些预定义字符集通常使用缩写形式，例如\d 等价于[0-9]。常用的预定义字符类如表 15-1 所示。

表 15-1　常用的预定义字符类

预定义字符	说　明
.	除行终止符以外的任何字符
\d	数字，等价于[0-9]
\D	非数字，等价于[^0-9]
\s	空白字符，等价于[\t\n\r\f\v]
\S	非空白字符，等价于[^\t\n\r\f\v]
\w	单词字符，等价于[a-zA-Z0-9_]
\W	非单词字符，等价于[^a-zA-Z0-9]

15.3.5　边界匹配符

字符串匹配往往涉及从某个位置开始匹配，例如行的开头或结尾、单词边界等。边界匹配符用于匹配字符串的位置如表 15-2 所示。

表 15-2　边界匹配符

边界匹配符	含　义	说　明
^	行开头	(1) "^a"匹配"abc"中的"a"，"^b"不匹配"abc"中的"b"；(2)"^\s*"匹配"　abc"中的左边空格
$	行结尾	(1) "c$"匹配"abc"中的"c"，"b$"不匹配"abc"中的"b"；(2)"^123$"匹配"123"中的"123"；(3)"\s*$"匹配"　abc "中的右边空格
\b	单词边界	r'\bfoo\b'匹配'foo'、'foo.'、'(foo)'、'bar foo baz'，但不匹配'foobar'或'foo3'
\B	非单词边界	r'py\B'匹配'python'、'py3'、'py2'，但不匹配'happpy'、'sleepy.'、'py!'
\A	字符串开头	和^的区别是：\A 只匹配整个字符串的开头，^匹配每一行开头
\Z	字符串结尾(除最后行终止符)	和$的区别是：\Z 只匹配整个字符串的结尾，$匹配每一行结尾

15.3.6　重复限定符

使用重复限定符可以指定重复的次数。例如中华人民共和国邮政编码由 6 位数字组成，使用重复限定符“\d{6}”表示数字字母重复 6 次。重复限定符如表 15-3 所示。

表 15-3 重复限定符

重复限定符	说　明
X?	X 重复 0 次或 1 次,等价于 X{0,1}。例如,"colou? r"可以匹配"color"或者"colour"
X*	X 重复 0 次或多次,等价于 X{0,}。例如,"zo*"可以匹配"z"、"zo"、"zoo"等
X+	X 重复 1 次或多次,等价于 X{1,}。例如,"zo+"可以匹配"zo"和"zoo",但不匹配"z"
X{n}	X 重复 n 次。例如\b[0-9]{3},匹配 000～999;"o{2}"不能与"Bob"中的"o"匹配,但是可以与"food"中的两个"o"匹配
X{n,}	至少重复 n 次。例如,"o{2,}"不匹配"Bob"中的"o",但是匹配"foooood"中所有的 o;"o{1,}"等价于"o+";"o{0,}"等价于"o*"
X{n,m}	重复 n 到 m 次。例如,"o{1,3}"匹配"fooooood"中的前 3 个 o;"o{0,1}"等价于"o?"

15.3.7 匹配算法:贪婪和懒惰

1. 贪婪性匹配算法

在默认情况下,Python 正则表达式的匹配算法采用贪婪性算法。例如:

```
>>> import re                                        #导入模块 re
>>> re.findall("<.+>", "<book><title>Python</title><author>Jiang<author></book>")
['<book><title>Python</title><author>Jiang<author></book>']
```

在该例中,正则表达式"<.+>"以<开始,紧跟 1 个或多个字符,以>结束,即 XML 的开始或结束标签,但结果并不是"<book>",这是因为 Python 正则表示匹配算法针对重复限定符,默认采用贪婪性匹配算法。

贪婪性匹配算法是指重复限定符会导致正则表达式引擎试图尽可能多地重复前导字符,只有当这种重复会引起整个正则表达式匹配失败时引擎才会进行回溯。

在该例中,正则表达式"<.+>"的第一个字符"<"为普通字符,匹配字符串的第一个"<";".+"匹配 1 个或多个字符(换行符除外),即匹配字符 b 并一直匹配其余的字符,直到换行符,匹配失败(.不匹配换行符)。于是引擎开始对下一个">"进行匹配,引擎会试图将">"与换行符进行匹配,结果失败了。于是引擎进行回溯,直到"<.+"匹配"<book><title>Python</title><author>Jiang<author></book",则">"与">"匹配。于是引擎找到了一个匹配"<book><title>Python</title><author>Jiang<author></book>"。

2. 懒惰性匹配算法

贪婪性算法返回了一个最左边的最长的匹配。如果在重复限定符后面加后缀"?",则正则表达式引擎使用懒惰性匹配算法,如表 15-4 所示。

表 15-4 懒惰性匹配算法

符　号	说　明
*?	重复任意次,但尽可能少重复
+?	重复 1 次或更多次,但尽可能少重复
??	重复 0 次或 1 次,但尽可能少重复
{n,m}?	重复 n 到 m 次,但尽可能少重复
{n,}?	重复 n 次以上,但尽可能少重复

例如:

```
>>> re.findall("<.+?>", "<book><title>Python</title><author>Jiang<author></book>")
```

```
['<book>', '<title>', '</title>', '<author>', '<author>', '</book>']
```

懒惰性匹配是指重复限定符会导致正则表达式引擎试图尽可能少地重复前导字符，只有当这种重复会引起整个正则表达式匹配失败时引擎才会进行回溯。

在该例中，正则表达式“<.+>”的第一个字符“<”为普通字符，匹配字符串的第一个“<”；".+"匹配1个或多个字符(换行符除外)，即匹配字符b；然后试图匹配“>”，匹配字符o失败，于是引擎回溯，“<.+”匹配“<bo”；然后试图匹配“>”，匹配字符o失败，于是引擎回溯，“<.+”匹配“<boo”；然后试图匹配“>”，匹配字符k失败，于是引擎回溯，“<.+”匹配“<book”，“>”匹配“>”。于是引擎找到了一个匹配“<book>”。

15.3.8　选择分支

在正则表达式中“|”表示选择，用于选择匹配多个可能的正则表达式中的一个。例如“red|green|blue”。

在正则表达式中，选择符“|”的优先级最低。如果需要，可以使用圆括号来限制选择符的作用范围。例如“\b(red|green|blue)\b>>”。

中国的电话号码一般为区号-电话号码，区号为3位或4位数字，电话号码为6位或8位数字，故其正则表达式为(0\d{2}|0\d{3})-(\d{8}|d{6})。例如：

```
>>> re.findall(r"((0\d{2}|0\d{3})-(\d{8}|d{6}))", "电话号码 021-62232333")
[('021-62232333', '021', '62232333')]
```

如果正则表达式包含组，则re.findall()返回组的列表。

15.3.9　分组和向后引用

1. 分组

重复限定符重复前导字符，如果需要重复多个字符，则需要把正则表达式的一部分放在圆括号()内，形成分组，然后对整个组使用诸如重复操作符的正则操作。

例如IP地址的一般形式为ddd.ddd.ddd.ddd，ddd.重复了3次，可以使用分组(\d{1,3}\.){3}\d{1,3}。

再如：

```
>>> re.findall("((\d{1,3}\.){3}\d{1,3})", "IP 地址 192.168.0.1")
                                        # 输出:[('192.168.0.1', '0.')]
```

2. 分组缓存和引用

当用“()”定义了一个正则表达式组后，正则引擎会把被匹配的组按照顺序编号，存入缓存。对被匹配的组可以进行向后引用：“\1”引用第一个匹配的组，“\2”引用第二个匹配的组，依此类推。“\0”则引用整个被匹配的正则表达式本身。

分组引用一般用于对称的模式，例如HTML的开始和结束标签。网页中包含开始标签、结束标签及中间文本，即<h1>News</h1>，可以使用正则表达式：

```
<([a-zA-Z][a-zA-Z0-9]*)[^>]*>.*?</\1>
```

首先，“<”匹配第一个字符“<”；然后[a-zA-Z]匹配h，[a-zA-Z0-9]*将会匹配0到多次字母数字，后面紧接着0到多个非“>”的字符；接着正则表达式的“>”将会匹配“<B>”的“>”。接下来正则引擎将对结束标签之前的字符进行惰性匹配，直到遇到一个“</”符号。然后正则表达式中的“\1”表示对前面匹配的组“([a-zA-Z][a-zA-Z0-9]*)”进行引用，引擎缓存的内容

为 h1，所以需要被匹配的结尾标签为“</h1 >”。例如：

```
>>> re.search(r"<([a-zA-Z][a-zA-Z0-9]*)[^>]*>.*?</\1>", r"abc<h1>News</h1>xyz")
<re.Match object; span=(3, 16), match='<h1>News</h1>'>
```

说明：

（1）可以多次引用组。例如“([a-b])x\1x\1”匹配“axaxa”和“bxbxb”。

（2）如果引用的组没有有效的匹配，则引用到的内容为空。

（3）引用不能用于其自身。例如“([a-c]\1)”是错误的。同样，“\0”表示正则表达式匹配本身，故只能用于替换操作中。

（4）引用不能用于字符集内部。例如“(a)[\1b]”中的“\1”被解释为八进制转码。

（5）向后引用会降低引擎的速度，因为它需要存储匹配的组。

（6）如果不需要向后引用，即对某个组不存储，可以使用组前缀“?:”。例如“Get(?:Value)”，“(”后面紧跟的“?:”告诉引擎对于组(Value)不存储。

（7）当对组使用重复操作符时，缓存里的引用内容会被不断刷新，只保留最后匹配的内容。例如，“([abc]+)=\1”将匹配“cab=cab”，但是“([abc])+=\1”不匹配“cab=cab”。因为([abc])第一次匹配 c 时，“\1”代表“c”；然后([abc])会继续匹配 a 和 b。最后“\1”代表“b”，所以它会匹配“cab=b”。

3. 分组命名和引用

在 Python 中对组可以进行如下命名并引用：

```
(?P<name>group)                       #组命名
(?P=name)                             #引用命名组
```

例如：

```
>>> m = re.search(r"(?P<Area>\d+)-(?P<No>\d+)", "电话号码:021-62232333")
>>> m.groupdict()                     #输出:{'Area': '021', 'No': '62232333'}
```

4. 分组的扩展语法

分组支持的扩展语法如表 15-5 所示。

表 15-5 分组扩展语法

符 号	说 明
(? aiLmsux)	应用于该组的匹配标志选项
(?: ...)	不存储组
(? P<name>...)	命名组
(? P=name)	引用命名组
(? #...)	注解。例如：“(? #comment)be”匹配 beg 中的 be
(? =...)	后置条件。例如：“be(?=ing)”匹配 being 中的 be，但不匹配 been 中的 be
(?!...)	后置非条件。例如：“be(?! ing)”不匹配 being 中的 be，但匹配 been 中的 be
(? <=...)	前置条件。例如：“(? <=pre)fix”匹配 prefix 中的 fix，但不匹配 suffix 中的 fix
(? <!...)	前置非条件。例如：“(? <! pre)fix”不匹配 prefix 中的 fix，但匹配 suffix 中的 fix

续表

<table>
<tr><th>符　　号</th><th>说　　明</th></tr>
<tr><td>(? (id/name)yes-pattern|no-pattern)</td><td>条件判别。如果组 id/name 存在，则 yes-pattern，否则 no-pattern。例如：(<)? (\w+@\w+(?: \.\w+)+)(? (1)>|$)匹配< user@host. com >和 user@host. com，但不匹配< user@host. com</td></tr>
</table>

15.3.10　正则表达式的匹配模式

正则表达式引擎都支持不同的匹配模式，也称之为匹配选项。例如，“/i”使正则表达式对大小写不敏感；“/m”开启多行模式，即“^”和“$”匹配新行符的前面和后面的位置。匹配模式可以通过分组扩展语法支持，即(? aiLmsux)，也可以通过匹配选项支持。

15.3.11　常用正则表达式

常用的正则表达式如表 15-6 所示。

表 15-6　常用的正则表达式

<table>
<tr><th>用　　途</th><th>正则表达式</th></tr>
<tr><td>Internet 电子邮件地址</td><td>^\w+([-+. ']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$</td></tr>
<tr><td>中华人民共和国电话号码</td><td>^(\(\d{3}\)|\d{3}-)? \d{8}$</td></tr>
<tr><td>中华人民共和国邮政编码</td><td>^\d{6}$</td></tr>
<tr><td>Internet URL</td><td>^https?://\w+(?: \.[^\.]+)+(?: /.+)*$</td></tr>
<tr><td>中华人民共和国身份证号码(ID 号)</td><td>^\d{17}[\d|X]|\d{15}$</td></tr>
</table>

15.4　正则表达式模块 re

15.4.1　匹配和搜索函数

正则表达式模块 re 提供了以下若干用于匹配和搜索的函数。

- re. match(pattern, string, flags=0)：若匹配，返回 Match 对象，否则返回 None。
- re. search(pattern, string, flags=0)：若匹配，返回 Match 对象，否则返回 None。
- re. findall(pattern, string, flags=0)：返回匹配结果列表；若 pattern 中包含组，返回组的列表。
- re. finditer(pattern, string, flags=0)：返回所有匹配结果(Match 对象)的迭代器。

其中，pattern 为匹配模式；string 为要匹配的字符串；flags 为匹配选项。

match()函数从字符串头部开始匹配，而 search()在字符串的任何位置匹配。例如：

```
>>> re.match("to", "To be,\nor not to be")                #无匹配
>>> re.search("^to", "To be,\nor not to be")              #无匹配
>>> re.search("to", "To be,\nor not to be")               #匹配
<_sre.SRE_Match object at 0x02D21528>
>>> re.findall("be", "To be,\nor not to be")              #输出:['be', 'be']
```

```
>>> re.finditer("be", "To be,\nor not to be")            #返回结果迭代器
<callable_iterator object at 0x02D22BF0>
>>> for i in re.finditer("be", "To be,\nor not to be"):print(i, end=' ')
<_sre.SRE_Match object at 0x02D21528> <_sre.SRE_Match object at 0x02D21608>
```

15.4.2 匹配选项

正则表达式模块 re 中包括表 15-7 所示的匹配选项。

表 15-7 re 模块的匹配选项

匹配选项	说明
re.A、re.ASCII	\w、\W、\b、\B、\d、\D、\s 以及\S,只匹配 ASCII 字符
re.I、re.IGNORECASE	忽略大小写。例如: `>>> re.match("to", "To be,\nor not to be")` #无匹配 `>>> re.match("to", "To be,\nor not to be", re.I)` #匹配 `<_sre.SRE_Match object at 0x02D21640>`
re.L、re.LOCALE	\w、\W、\b、\B、\s 以及\S,与本地字符集有关
re.M、re.MULTILINE	多行匹配模式。例如: `>>> re.search("^or", "To be,\nor not to be")` #无匹配 `>>> re.search("^or", "To be,\nor not to be",re.M)` #匹配 `<_sre.SRE_Match object at 0x02D21528>`
re.S、re.DOTALL	匹配所有字符,包括换行符
re.X、re.VERBOSE	忽略空格并可以使用#注释,提高可读性。例如: `b = re.compile(r"\d+\.\d*")` #等价于 `a = re.compile(r"""\d +` #整数部分 `\.` #小数点 `\d*` #小数部分""", re.X)
re.DEBUG	输出调试信息

15.4.3 正则表达式对象

使用正则表达式模块 re 中的 re.compile()函数可以将正则表达式编译为正则表达式对象 regex,然后使用其对象方法处理字符串。

- regex=re.compile(pattern, flags=0):编译模式,生成正则表达式对象。
- regex.search(string[, pos[, endpos]]):若匹配,返回 Match 对象,否则返回 None。
- regex.match(string[, pos[, endpos]]):若匹配,返回 Match 对象,否则返回 None。
- regex.findall(string[, pos[, endpos]]):返回匹配结果列表;若 pattern 中包含组,返回组的列表。
- regex.finditer(string[, pos[, endpos]]):返回所有匹配结果(Match 对象)的迭代器。

其中,pattern 为匹配模式;string 为要匹配的字符串;flags 为匹配选项;pos 和 endpos 为搜索范围:pos~endpos−1。

正则表达式对象方法 search()、match()、findall()和 finditer()与 re 模块中的对应函数基本一致。例如:

```
>>> regex1 = re.compile('to')
>>> regex2 = re.compile('^to')
```

```
>>> regex1.match("To be,\nor not to be")           #无匹配
>>> regex2.search("To be,\nor not to be")          #无匹配
>>> regex1.search("To be,\nor not to be")          #匹配
<_sre.SRE_Match object at 0x02C73528>
>>> regex1.findall("To be,\nor not to be")         #返回结果列表:['to']
>>> regex1.finditer("To be,\nor not to be")        #返回结果迭代器
<callable_iterator object at 0x02C26730>
```

15.4.4　匹配对象

使用正则表达式模块 re 中的函数 search()和 match(),或者正则表达式对象方法 search()和 match(),返回的结果为匹配对象(Match object)。使用匹配对象的方法可以进行匹配结果处理。

- m.group([group1, …]):返回匹配的 1 个或多个组。
- m.groups(default=None):返回匹配的所有子组,结果为元组。
- m.groupdict(default=None):返回匹配的所有命名组,结果为字典。
- m.start([group]):返回匹配的组的开始位置。
- m.end([group]):返回匹配的组的结束位置。
- m.span([group]):返回匹配的组的位置范围,即(m.start(group), m.end(group))。

例如:

```
>>> m = re.search("be", "To be,\nor not to be")
>>> m.group()                                      #输出:'be'
>>> m.span()                                       #输出:(3, 5)
```

再如:

```
>>> m = re.search(r"(?P<Area>\d+)-(?P<No>\d+)", "电话号码:021-62232333")
>>> m.group(), m.group(0), m.group(1), m.group(2)
('021-62232333', '021-62232333', '021', '62232333')
>>> m.groups()                                     #输出:('021', '62232333')
>>> m.groupdict()                                  #输出:{'Area': '021', 'No': '62232333'}
```

15.4.5　匹配和替换

使用正则表达式模块 re 中的函数 sub()和 subn(),或者正则表达式对象方法 sub()和 subn(),可以使用正则表达式匹配字符串,用指定内容替换结果,并返回替换后的字符串。其形式如下:

- re.sub(pattern, repl, string, count=0, flags=0):返回替换后的字符串。
- re.subn(pattern, repl, string, count=0, flags=0):返回元组,即(替换后的字符串,替换次数)。
- regex.sub(repl, string, count=0):同 re.sub()。
- regex.subn(repl, string, count=0):同 re.subn()。

其中,pattern 为匹配模式;string 为要匹配和替换的字符串;repl 为要替换的内容;count 为替换的最大次数;flags 为匹配选项。例如:

```
>>> re.sub('bad', 'good', 'It tastes bad.')        #输出:'It tastes good.'
```

在编辑文字时很容易会输入重复单词,例如"thethe"。使用<<\b(\w+)\s+\1\b>>可以检测到这些重复单词。如果要删除第二个单词,只要简单地利用替换功能替换掉"\1"就可

以了。

15.4.6 分隔字符串

使用正则表达式模块 re 中的函数 split(),或正则表达式对象方法 split(),可以使用正则表达式匹配字符串(匹配分隔符),并分隔字符串,返回分隔后的字符串列表。其形式如下:

- **re.split(pattern, string, maxsplit = 0, flags = 0):** #返回分隔后的字符串列表
- **regex.split(string, maxsplit = 0):** #同 re.split()

其中,pattern 为匹配模式;string 为要匹配和分隔的字符串;maxsplit 为分隔的最大次数;flags 为匹配选项。例如:

```
>>> re.split('\W+', 'Good, better, best!')         #'\W+'1 个以上非单词字符,输出:['Good', 'better',
                                                   #'best', '']
>>> re.split('\W+', 'Good, better, best!', 1)      #分隔 1 次,输出:['Good', 'better, best!']
>>> re.split('(\W+)', 'Good, better, best!')       #使用分组时同时返回分隔字符
['Good', ', ', 'better', ', ', 'best', '!', '']
>>> re.split('\d', '1a2b3c4d')                     #分隔符为数字字符,输出:['', 'a', 'b', 'c', 'd']
```

15.5 正则表达式应用举例

15.5.1 字符串包含验证

通过正则表达式模块 re 的匹配和搜索,或者正则表达式对象的匹配和搜索,可以验证一个字符串是否与指定模式匹配,即实现字符串包含验证。

【例 15.10】 验证一个字符串是否为有效的电子邮件格式(emailaddress_check.py)。

```
import os, re
def check_email(strEmail):
    regex_email = re.compile(r'^[\w\.\-]+@([\w\-]+\.)+[\w\-]+$')
    result = True if regex_email.match(strEmail) else False
    return result
#测试代码
if __name__ == '__main__':
    str1 = "hjiang@yahoo.com"                    #有效的电子邮箱
    str2 = "hjiang.yahoo.com"                    #无效的电子邮箱
    print(str1,'是有效的电子邮件格式吗?', check_email(str1))
    print(str2,'是有效的电子邮件格式吗?', check_email(str2))
```

程序运行结果如下。

```
hjiang@yahoo.com 是有效的电子邮件格式吗? True
hjiang.yahoo.com 是有效的电子邮件格式吗? False
```

15.5.2 字符串的查找和拆分

使用正则表达式模块 re 中的函数 split()或正则表达式对象方法 split()可以分割字符串;也可以通过 re 中的函数 findall()或正则表达式对象方法 findall()分割字符串,因为如果匹配中包含组,findall 将返回组的列表。

【例 15.11】 根据给定正则表达式的匹配拆分字符串示例(tasklist.py)。该例中使用了 os.popen 执行操作系统命令并返回管道(管道可以参见本书的第 6.6.3 节)。

```
import os, re
def tasklist():
    regex_task = re.compile(r'([\w.]+(?:[\w.]+)*)\s\s+(\d+) \w+\s\s+\d+\s\s+
([\d,]+ K)')
    with os.popen('tasklist /nh', 'r') as f:
        for line in f:
            print(regex_task.findall(line.strip()))
if __name__ == '__main__':
    tasklist()
```

程序运行结果如下。

```
[]
[('System Idle Process', '0', '24 K')]
[('System', '4', '7,660 K')]
[('smss.exe', '328', '1,212 K')]
[('csrss.exe', '496', '5,328 K')]
…
```

15.5.3 字符串的替换和清除

使用正则表达式模块 re 中的函数 sub()或正则表达式对象方法 sub()可以实现字符串替换。例如把 HTML 文件的所有大写 HTML 标记替换成小写标记。

【例 15.12】 从输入字符串中清除 HTML 标记(html_txt.py)。

```
import re
def html_txt(htmlwithtag):
    regex_href = re.compile(r'<.+?>')
    return regex_href.sub('', htmlwithtag)     #替换为空'',并返回替换结果
#测试代码
if __name__ == '__main__':
    htmltext = r'<a href=\"index.html\">Welcome to Python world!</a>'
    print(html_txt(htmltext))
```

程序运行结果如下。

```
Welcome to Python world!
```

15.5.4 字符串的查找和截取

通过正则表达式对象的 findall()/finditer()方法可以查找与该模式匹配的结果列表。例如从网页中查找所有的 URL。

【例 15.13】 从指定的网页中查找所有的超链接地址(url_extract.py)。该例中使用了 urllib.request.urlopen 打开网页(具体请参见本书的第 18.3 节)。

```
import re, urllib.request
def url_extract(homepage):
    regex_href = re.compile(r'href="(.+?)"')
    f = urllib.request.urlopen(homepage)
    webcontents = f.read().decode()
    matches = regex_href.finditer(webcontents)
    for m in matches:
        print(m.group(1))
#测试代码
if __name__ == '__main__':
```

```
    www = r'http://www.baidu.com'
    url_extract(www)
```

程序运行结果如下。

```
http://www.nuomi.com
http://news.baidu.com
http://www.hao123.com
http://map.baidu.com
...
```

15.6 应用举例

15.6.1 文本统计

文本统计程序可以从文本文件中读取字符串序列,统计文本中包含的段落数、行数、句数、单词数,以及统计各单词出现的频率。

频率计数广泛用于从海量的数据中统计各种事件出现的频率。例如,语言学家从文章中发现单词的使用模式,商人从订单中发现重要的客户,等等。

【例 15.14】 文本统计示例程序(text_stat.py)。

```
import re
import collections
def analyze_text(text):
    paragraphs = re.split("\n\n", text)
    paragraph_count = len(paragraphs)
    print("段落数:{0}".format(paragraph_count))
    lines = re.split("\n", text)
    line_count = len(lines)
    print("行数:{0}".format(line_count))
    sentences = re.split("[.?!]", text)
    sentence_count = len(sentences)
    print("句数:{0}".format(sentence_count))
    words = re.split(r"\W+", text)
    word_count = len(words)
    print("单词数:{0}".format(word_count))
    freqs = collections.Counter(words)
    print("频率最高的 10 个单词:")
    for (w, n) in freqs.most_common(10):
        print("{0:10}:{1:10}".format(w, n))

if __name__ == "__main__":
    filename = "tomsawyer.txt"
    with open(filename,"r") as f:
        text = f.read()
    analyze_text(text.strip())
```

程序运行结果如下。

```
段落数:2
行数:3
句数:12
单词数:240
频率最高的 10 个单词:
Tom       :        10
```

```
s       :       10
and     :       10
of      :       9
the     :       8
his     :       7
is      :       7
in      :       6
a       :       5
Huck    :       5
```

15.6.2　基因预测

基因是生命的本质，生物学家用字母 A、C、T 和 G 分别代表生物体 DNA 的 4 个碱基。基因由一序列的密码子组成，每个密码子是一个由一系列代表氨基酸的 3 个碱基组成的序列。

判断某字符串是否对应一个潜在的基因的准则如下：

(1) 基因长度为 3 的倍数。

(2) 以 ATG 标识基因的开始，以 TAG、TAA 或者 TGA 标识基因的结束。

(3) 除了结束部分，中间部分不包括 TAG、TAA 或者 TGA。

编写程序，查找输出定义文件中行字符串为潜在基因的思维和流程如下：

(1) 以文本方式打开文件。

(2) 循环读取各行内容，判断是否为潜在基因，如果是，输出行号和行的内容。

【例 15.15】　基因预测示例程序(gene_scan.py)。

```
def isPotentialGene(dna):
    #基因长度为 3 的倍数，否则返回 Fasle
    if len(line) % 3 != 0:
        return False
    #基因以 ATG 开始，否则返回 Fasle
    if not dna.startswith('ATG'):
        return False
    #基因以 TAG、TAA 或者 TGA 结束，否则返回 Fasle
    if dna[-3:] not in ('TAG', 'TAA', 'TGA'):
        return False
    #基因中间部分不包括密码子 TAG、TAA 或者 TGA,否则返回 False
    for i in range(3,len(dna)-3,3):
        if dna[i:i+3] in ('TAG', 'TAA', 'TGA'):
            return False
    return True
if __name__ == "__main__":
    filename = "gene.txt"
    for lineno, line in enumerate(open(filename,"r")):
        if isPotentialGene(line.strip()):
            print("{0}:{1}".format(lineno+1, line.strip()))
```

程序运行结果如下。

```
1:ATGCGCCTGCGTCTGTATAG
3:ATGCGCCTGCGTCTGTATAA
4:ATGCGCCTGCGTCTGTATGA
```

15.6.3　字符串的简单加密和解密

基于按位逻辑异或的简单加密算法的原理如下：给定明文字符(例如 A)和秘钥字符(例

如 P),其对应的 ASCII 码的按位逻辑异或即是加密后的密文字符,密文字符的 ASCII 码与相同秘钥字符的 ASCII 码为加密前的明文。例如:

```
>>> ord('A') ^ ord('P')          #输出:17
>>> chr(17 ^ ord('P'))           #输出:'A'
```

故基于按位逻辑异或的简单字符串加密算法和解密算法可以共用一个函数,其设计思路如下:

(1) 给定字符串 text(例如 The quick brown fox jumps over the lazy dog)和 key(例如 Python_1),使用 itertools.cycle(key)构造一个循环字符串迭代器 keys。

(2) 循环处理 text 的每个字符,使用 keys 中的对应字符进行按位逻辑异或运算,结果就是加密后的密文(如果解密,结果就是解密后的明文)。

【例 15.16】 字符串简单加密和解密(crypt.py)。

```
from itertools import cycle
def crypt(text, key):
    result = []
    keys = cycle(key)
    for ch in text:
        result.append(chr(ord(ch)^ord(next(keys))))
    return ''.join(result)
#测试代码
if __name__ == '__main__':
    plain = 'The quick brown fox jumps over the lazy dog'
    key = 'Python_1'
    print('加密前明文:{}'.format(plain))
    encrypted = crypt(plain, key)
    print('加密后密文:{}'.format(encrypted))
    decrypted = crypt(encrypted, key)
    print('解密后明文:{}'.format(decrypted))
```

程序运行结果如下。

```
加密前明文: The quick brown fox jumps over the lazy dog
加密后密文: □□□H□6R;Y□□ □1□6□↑H□□2A#Y□
 E8T□□□&□4□□
解密后明文: The quick brown fox jumps over the lazy dog
```

15.7 复 习 题

一、填空题

1. Python 语句序列“s1='red hat'; print(str.upper(s1))”的运行结果是________,str.swapcase(s1)的结果是________,s1.title()的结果是________,s1.replace('hat', 'cat')的结果是________。

2. 在 Python 中,设有 s='abc',则 s.zfill(7)、s.center(7, ' ')、s.ljust(7)、s.rjust(7, '0')的结果分别为________。

3. 在 Python 中,设有 s='a,b,c'、s2=('x','y','z')以及 s3=':',则 s.split(',')、s.rsplit(',', 1)、s.partition(',')、s.rpartition(',')、s3.join('abc')、s3.join(s2)的结果分别为________。

4. Python 语句 re.match('back', 'text.back')的执行结果是________。

5. Python 语句 re.findall("to", "Tom likes to swim too")的执行结果是________。

6. Python 语句 re.findall("bo[xy]", "The boy is sitting on the box")的执行结果是________。

7. 中华人民共和国邮政编码由 6 位数字组成,使用重复限定符________表示数字字母重复 6 次。

8. 正则表达式引擎均支持不同的匹配模式,也称之为匹配选项,其中,________使正则表达式对大小写不敏感,________开启多行模式。

9. Python 语句 re.sub('hard', 'easy', 'Python is hard to learn.')的执行结果是________。

10. Python 语句 re.split('\W+', 'go, went, gone')的执行结果是________。

11. Python 语句 re.split('\d', 'a1b2c3')的执行结果是________。

二、思考题

1. 在用 Python 匹配 HTML tag 时,<.*>和<.*?>有什么区别?

2. Python 中的 search()和 match()有什么区别?

3. 如何使用 Python 查询和替换一个文本字符串?

4. 正则表达式包括哪些元字符?

5. 阅读下面的 Python 语句,请问输出结果是什么?

```
fruits = ['pear', 'apple', 'kiwi', 'avocado', 'orange']
print("\n".join(fruits))
```

6. 阅读下面的 Python 语句,请问输出结果是什么?

```
name = "happy birthday"
print("%s" % name[6:11])
name1 = name.replace(name[6],'B')
print(name1)
```

7. 下列 Python 语句的输出结果是________。

```
import re; sum = 0; pattern = 'boy'
if re.match(pattern, 'boy and girl'): sum += 1
if re.match(pattern, 'girl and boy'): sum += 2
if re.search(pattern, 'boy and girl'): sum += 3
if re.search(pattern, 'girl and boy'): sum += 4
print(sum)
```

8. 下列 Python 语句的输出结果是________。

```
import re; re.match("to", "Tom likes to swim too")
re.search("to", "Tom likes to swim too");
re.findall("to", "Tom likes to swim too")
```

9. 下列 Python 语句的输出结果是________。

```
import re; m = re.search("to", "Tom likes to swim too")
print(m.group(),m.span())
```

10. 请使用各种方法判断字符变量 c 是否为字母字符(不区分大小写字母)。

11. 请使用各种方法判断字符变量 c 是否为数字字符。

12. 请使用各种方法判断字符变量 c 是否为大写字母。

13. 请使用各种方法判断字符变量 c 是否为小写字母。

15.8 上机实践

1. 完成本章中的例 15.1～例 15.16,熟悉 Python 语言中的字符串和文本处理程序设计。

2. 统计输入的字符串中英文字母、数字、空格和其他字符出现的次数,程序运行效果如图 15-1 所示。

```
请输入字符串：This is   a test. 123  45678~ End?
所有字母的总数为：    33
英文字母出现的次数： 13
数字出现的次数：     8
空格出现的次数：     9
其他字符出现的次数： 3
```

图 15-1 统计字符运行效果

提示:

本实践题使用表 15-8 中的字符串对象的方法和 str 类方法确定字符/字符串是否为字母、数字、空格等。

表 15-8 本实践题所使用的字符串对象的方法/str 类方法

方 法	功 能
isalpha()	判断字符/字符串是否全为字母
isdigit()	判断字符/字符串是否全为数字(0～9)
isspace()	判断字符/字符串是否只包含空白字符

3. 编写程序,分别输入 3 个字符串,依次验证其是否为有效的中华人民共和国电话号码、中华人民共和国邮政编码和网站网址格式,运行效果如图 15-2 所示。

```
请输入中国电话号码: 021-12345678
021-12345678 是有效的电话号码格式吗? True
请输入中国邮政编码: 200062
200062 是有效的邮政编码格式吗? True
请输入网站网址: http://www.ibm.com
http://www.ibm.com 是有效的网站网址格式吗? True
```

(a) 有效格式

```
请输入中国电话号码: 123456789
123456789 是有效的电话号码格式吗? False
请输入中国邮政编码: 123456789
123456789 是有效的邮政编码格式吗? False
请输入网站网址: http@ecnu.edu.cn
http@ecnu.edu.cn 是有效的网站网址格式吗? False
```

(b) 无效格式

图 15-2 验证有效格式运行效果

提示:

(1) 中华人民共和国电话号码(电话号码必须是 8 位号码,如果有区号,区号必须是 3 位)的正则表达式为^(\(\d{3}\)|\d{3}－)? \d{8} $。

(2) 中华人民共和国邮政编码(必须 6 位数字)的正则表达式为^\d{6} $。

(3) 网站网址(Internet URL)的正则表达式为^https?://\w＋(?: \.[^\.]＋)＋(?: /.＋)＊ $。

(4) 验证函数的参考代码如图 15-3 所示。

```
def check_phone(strPhone): #中华人民共和国电话号码
    regex_phone = re.compile(r'^(\(\d{3}\)|\d{3}-)?\d{8}$')
    result = True if regex_phone.match(strPhone) else False
    return result
def check_ZIP(strZIP):      #中华人民共和国邮政编码
    regex_ZIP = re.compile(r'^\d{6}$')
    result = True if regex_ZIP.match(strZIP) else False
    return result
def check_URL(strURL):      #网站网址
    regex_URL = re.compile(r'^https?://\w+(?:\.[^\.]+)+(?:/.+)*$')
    result = True if regex_URL.match(strURL) else False
    return result
```

图 15-3 验证有效格式参考代码

15.9　案例研究：NLTK 与自然语言处理

NLTK(Natural Language ToolKit)是一套基于 Python 的自然语言处理工具集。

本章案例研究通过 Python 自然语言分析处理库 nltk 的安装和基本使用帮助读者了解自然语言处理的基本方法。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 16 章

文件和数据交换



视频讲解

应用程序往往需要从磁盘文件中读取数据，或者把数据存储到磁盘文件中，以持久地保存应用程序的数据。

16.1　文件操作相关模块概述

Python 标准库中包括下列文件处理的相关模块。

- io 模块：文件流的输入/输出操作模块。
- bz2 模块：读取和写入基于 bzip2 压缩算法的压缩文件。
- gzip 模块：读取和写入基于 gzip 压缩算法的压缩文件。
- zipfile 模块：读取和写入基于 zip 压缩算法的压缩文件。
- zlib 模块：读取和写入基于 zlib 压缩算法的压缩文件。
- tarfile 模块：读取和写入 TAR 格式的卷文件(支持压缩和非压缩)。
- glob 模块：查找符合特定规则的文件路径名。
- fnmatch 模块：使用模式来匹配文件路径名。
- fileinput 模块：处理一个或多个输入文件。
- filecmp 模块：用于文件的比较。
- csv 模块：读取和写入 CSV 格式的文件。
- pickle 和 cPickle：序列号 Python 对象。
- xml 包：XML 核心处理操作。
- os 模块：基本操作系统功能，包括文件操作。
- json 模块：JSON 格式数据操作。

16.2　文本文件的读取和写入

在使用 open()函数打开或创建一个文件时，其默认的打开模式为只读文本文件。文本文件用于储存文本字符串，默认编码为 Unicode。

16.2.1　文本文件的写入

文本文件的写入一般包括 3 个步骤，即打开文件、写入数据和关闭文件。

1. 创建或打开文件对象

通过内置函数 open()可以创建或打开文件对象，并且可以指定覆盖模式(文件存在时)、

编码和缓存大小。例如：

```
f1 = open('data1.txt', 'w')         #创建或打开 data1.txt
f2 = open('data2.txt', 'x')         #创建文件 data2.txt,若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.txt', 'a')         #创建或打开 data1.txt,附加模式
```

2. 把字符串写入文本文件

在打开文件后，可以使用其实例方法 write()/writelines()把字符串写入文本文件，还可以使用实例方法 flush()强制把缓冲的数据更新到文件中。

- f.write(s)：把字符串 s 写入文件 f。
- f.writelines(lines)：依次把列表 lines 中的各字符串写入文件 f。
- f.flush()：把缓冲的数据更新到文件中。

实例方法 write()/writelines()不会添加换行符，但可以通过添加“\n”实现换行。例如：

```
f.write('123\n')                    #写入字符串,并换行
f.write('abc\n')                    #写入字符串,并换行
f.writelines(['456\n', 'def\n'])    #写入字符串,并换行
```

3. 关闭文件

在写入文件完成后，应该使用 close()方法关闭流，以释放资源，并把缓冲的数据更新到文件中。

```
f.close()                           #关闭文件
```

同时可以使用异常处理的 finally 子句，以保证即使发生异常也会关闭打开的文件。

```
f = open('data1.txt', 'w')          #打开文件
try:                                #文件处理操作
finally:
    f.close()                       #关闭文件
```

通常，文件操作一般采用 with 语句，以保证系统自动关闭打开的流。

```
with open('data1.txt', 'w') as f: #文件处理操作
```

【例 16.1】　文本文件的写入示例(textwrite.py)。

```
with open(r'c:\pythonpa\data1.txt', 'w') as f:
    f.write('123\n')                    #写入字符串
    f.write('abc\n')                    #写入字符串
    f.writelines(['456\n', 'def\n'])    #写入字符串
```

16.2.2　文本文件的读取

文本文件的读取一般包括 3 个步骤，即打开文件、读取数据和关闭文件。

1. 打开文件对象

通过内置函数 open()可以打开文件对象，并且可以指定编码和缓存大小。例如：

```
f1 = open('data1.txt', 'r')         #打开 data1.txt,若文件不存在,则导致 FileNotFoundError
```

2. 从打开的文本文件中读取字符数据

在打开文件后，可以使用下列实例方法读取字符数据。

- f.read()：从 f 中读取剩余内容直至文件结尾，返回一个字符串。
- f.read(n)：从 f 中最多读取 n 个字符，返回一个字符串；如果 n 为负数或 None，读取直至文件结尾。

- f.readline():从 f 中读取一行内容,返回一个字符串。
- f.readlines() :从 f 中读取剩余的多行内容,返回一个列表。

例如:

```
>>> f1 = open(r'c:\pythonpa\data1.txt', 'r')   #打开文件
>>> f1.readline()                              #读入一行内容,输出:'123\n'
>>> f1.readlines()                           #读入剩余的多行内容,输出:['abc\n', '456\n', 'def\n']
```

另外,文件可以直接迭代。文本文件按行迭代。例如:

```
>>> f1 = open(r'c:\pythonpa\data1.txt', 'r')
>>> for s in f1:
    print(s, end = '')
123
abc
456
def
```

3. 关闭文件

用户可以使用 close()方法关闭流,以释放资源。通常采用 with 语句,以保证系统自动关闭打开的流。

【例 16.2】 文本文件的读取示例(textread.py)。

```
with open(r'c:\pythonpa\data1.txt', 'r') as f:
    for s in f.readlines():
        print(s, end = '')
```

16.2.3 文本文件的编码

文本文件用于存储编码的字符串,在使用 open()函数打开文本文件时,可以指定所使用的编码,函数形式如下:

```
open(file, mode = 'r', buffering = -1, encoding = None, errors = None, newline = None, closefd = True, opener = None)
```

encoding 默认为 None,即不指定。默认的编码与平台有关,其值为:

```
>>> import sys
>>> sys.getdefaultencoding()                   #输出:'utf-8'
```

Python 内置的编码包括 utf-8、utf8、latin-1、latin1、iso-8859-1、mbcs(仅 Windows 系统)、ascii、utf-16、utf-32 等。例如:

```
>>> f = open("1.txt", mode = "w", encoding = "utf-8")
```

16.3 二进制文件的读取和写入

在使用 open()函数打开或创建一个文件时可以指定打开模式为'b',以打开二进制文件。文本文件用于存储编码的字符串,二进制文件直接存储字节码,被广泛用于存储各种程序数据,例如图像文件、音频/视频文件等。

16.3.1 二进制文件的写入

二进制文件的写入一般包括 3 个步骤，即打开文件、写入数据和关闭文件。

1. 创建或打开文件对象

通过内置函数 open()，指定打开模式为'b'，可以创建或打开二进制文件对象，并且可以指定覆盖模式（文件存在时）和缓存大小。例如：

```
f1 = open('data1.dat', 'wb')   #创建或打开 data1.dat
f2 = open('data2.dat', 'xb')   #创建文件 data2.dat,若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.dat', 'ab')   #创建或打开 data1.dat,附加模式
```

2. 写入字节数据到二进制文件

在打开文件后，可以使用其实例方法 write()，写入字节数据（bytes 或 bytearray）到二进制文件，还可以使用实例方法 flush()强制把缓冲的数据更新到文件中。相关命令如下：

- **f.write(b)**　　　#将字节数据 b 写入到二进制文件 f，返回实际写入的字节数
- **f.flush()**　　　#将缓冲的数据更新到文件中

例如：

```
f1.write(b'123')          #写入字节数据
f1.write(b'abc')          #写入字节数据
```

3. 关闭文件

用户可以使用 close()方法关闭流，以释放资源。通常采用 with 语句，以保证系统自动关闭打开的流。

【例 16.3】 二进制文件的写入示例（binarywrite.py）。

```
with open(r'c:\pythonpa\data1.dat', 'wb') as f:
    f.write(b'123')       #写入字节数据
    f.write(b'abc')       #写入字节数据
```

16.3.2 二进制文件的读取

二进制文本文件的读取一般包括 3 个步骤，即打开文件、读取数据和关闭文件。

1. 打开文件对象

通过内置函数 open()，指定打开模式为'b'，可以打开二进制文件对象，并且可以指定编码和缓存大小。例如：

```
f1 = open('data1.dat', 'rb')   #打开 data1.dat,若文件不存在,则导致 FileNotFoundError
```

2. 从打开的文本文件中读取字符数据

在打开文件后，可以使用下列实例方法读取字符数据：

- f.read()：从 f 中读取剩余内容直至文件结尾，返回一个 bytes 对象。
- f.read(n)：从 f 中最多读取 n 个字节，返回一个 bytes 对象；如果 n 为负数或 None，读取直至文件结尾。
- f.readinto(b)：从 f 中最多读取 len(b)个字节到 bytes 对象 b。

例如：

```
f1.read(1)               #读取一个字节,结果:b'1'
f1.read()                #读取剩余字节,结果:b'23abc'
```

3. 关闭文件

用户可以使用 close 方法关闭流,以释放资源。通常采用 with 语句,以保证系统自动关闭打开的流。

【例 16.4】 二进制文件的读取示例(binaryread.py)。

```
with open(r'c:\pythonpa\data1.dat', 'rb') as f:
    b = f.read()
    print(b)
```

程序运行结果如下。

```
b'123abc'
```

16.4 随机文件访问

文件的写入和读取一般从当前位置开始(打开文件时位置为 0),直至文件结尾(EOF),即按顺序访问。文件对象支持 seek()方法,seek()通过字节偏移量将读取/写入位置移动到文件中的任意位置,从而实现文件的随机访问。seek()方法的命令形式如下:

```
seek(offset, whence = os.SEEK_SET)
```

其中,offset 为移动的字节偏移量,whence 为相对参考点(文件开始、当前位置、结尾,分别对应于 os.SEEK_SET、os.SEEK_CUR、os.SEEK_END,或 0、1、2)。

随机文件访问一般针对二进制文件,因为其存储内容为字节码。文本文件也可以使用 seek()方法,但多字节的偏移量不容易控制,有时候会导致无意义。

1. 创建或打开随机文件

随机文件一般同时提供读写操作,即使用内置函数 open()指定打开模式为'+'。例如:

```
f1 = open('data1.dat', 'w+b')   #创建或打开 data1.dat
f2 = open('data2.dat', 'x+b')   #创建文件 data2.dat,若 data2.txt 已存在,则导致 FileExistsError
f3 = open('data1.dat', 'a+b')   #创建或打开 data1.dat,附加模式
f4 = open('data1.dat', 'wb+')   #创建或打开 data1.dat,同'w+b'
```

2. 定位

在打开文件后,可以使用其实例方法 seek()进行定位,即将该文件的当前位置设置为给定值。例如:

```
f1.seek(0)                      #定位到开始位置
```

3. 写入/读取数据

打开文件并定位文件位置后,可以使用其实例方法 write()/read()写入或读取字节数据。例如:

```
f1.seek(0, os.SEEK_END)         #定位到结束位置
f1.write(b'hello')              #写入字节数据
f1.seek(3)                      #定位到第 3 个位置
f1.read(3)                      #读取 3 个字节,结果:b'abc'
```

4. 关闭文件

用户可以使用 close()方法关闭流,以释放资源。通常采用 with 语句,以保证系统自动关闭打开的流。

【例 16.5】 随机文件的读写示例(randomfile.py)。

```
import os
f = open('data.dat', 'w+b')          # 创建或打开文件 data.dat
f.seek(0)                            # 定位到开始位置
f.write(b'Hello')                    # 写入字节数据
f.write(b'World')                    # 写入字节数据
f.seek(-5, os.SEEK_END)              # 定位到结束位置的倒数第 5 个位置
b = f.read(5)                        # 读取 5 个字节
print(b)                             # 输出:b'World'
```

程序运行结果如下。

```
b'World'
```

16.5 内存文件的操作

在程序设计过程中,有时候需要在内存中创建临时文件,以实现数据的读写。Python 标准库模块 io 中的 StringIO 和 BytesIO 对象用于实现内存文本文件的操作和内存二进制文件的操作。

16.5.1 StringIO 和内存文本文件的操作

StringIO 实现了内存文本文件的读取操作,常用于字符串的缓存。StringIO 与文件操作的接口保持一致,包括 read()、readline()、readlines()、write()、writelines()等,即同样的文本文件操作代码可以用于 StringIO 操作。

创建内存文本文件的语法如下:

```
from io import StringIO
f = StringIO(s)                      # 基于可选的字符串 s 创建内存文本文件对象
```

在内存文本文件创建之后,可以使用内存文件对象 f 的 read()、readline()、readlines()、write()、writelines()等方法实现各种读写操作。

【例 16.6】 内存文本文件的读写示例(siofile.py)。

```
from io import StringIO
f = StringIO('Hello!\nHi!\nGoodbye!')
for s in f:
    print(s.strip())
```

程序运行结果如下。

```
Hello!
Hi!
Goodbye!
```

16.5.2 BytesIO 和内存二进制文件的操作

BytesIO 实现了内存二进制文件的读取操作,常用于字节码的缓存。BytesIO 与文件操作的接口保持一致,包括 read()、write()等,即同样的二进制文件操作代码可以用于 BytesIO 操作。

创建内存二进制文件的语法如下：

```
from io import BytesIO
f = BytesIO(b)                    #基于可选的字节码序列创建内存二进制文件对象
```

在内存二进制文件创建之后，可以使用内存文件对象 f 的 read()、write()等方法实现各种读写操作。

【例 16.7】 内存二进制文件的读写示例(biofile.py)。

```
from io import BytesIO
f = BytesIO()
f.write('中文'.encode('utf-8'))
f.seek(0)                         #定位到开始位置
b=f.read()                        #读取文件内容
print(b)                          #显示文件内容
print(f.getvalue())               #显示文件内容
```

程序运行结果如下。

```
b'\xe4\xb8\xad\xe6\x96\x87'
b'\xe4\xb8\xad\xe6\x96\x87'
```

16.6 文件的压缩和解压缩

gzip 和 bz2 模块实现了用于 gzip 和 bz2 格式压缩文件的 open()函数，支持打开对应格式的压缩文件，从而实现压缩文件的读取和写入处理。

打开压缩文件的语法如下：

```
f = gzip.open()                   #其语法格式与内置函数 open()类似
f = bz2.open()                    #其语法格式与内置函数 open()类似
```

在压缩文件打开之后，可以使用文件对象 f 的 read()、readline()、readlines()、write()、writelines()等方法实现各种读写操作。

【例 16.8】 使用 gzip 模块压缩和解压缩文件的示例(gzipfile.py)。

```
import sys, gzip
filename = sys.argv[0]            #Python 文件名，即 gzipfile.py
filenamezip = filename + '.gz'  #文件扩展名为.gz
#gzip 压缩
with gzip.open(filenamezip, 'wt') as f:
    for s in open(filename, 'r'):
        f.write(s)
#gzip 解压缩/提取
for s in gzip.open(filenamezip, 'r'):
    print(s)
```

16.7 CSV 格式文件的读取和写入

CSV 是逗号分隔符文本格式，常用于 Excel 和数据库中数据的导入和导出。Python 标准库模块 csv 提供了读取和写入 CSV 格式文件的对象。

本节基于以下 scores.csv 文件，其内容为：

```
学号,姓名,性别,班级,语文,数学,英语
```

```
101511,宋颐园,男,一班,72,85,82
101513,王二丫,女,一班,75,82,51
101531,董再永,男,三班,55,74,79
101521,陈香燕,女,二班,80,86,68
101535,周一萍,女,三班,72,76,72
```

16.7.1　csv.reader 对象和 CSV 文件的读取

csv.reader 对象用于从 CSV 文件读取数据(格式为列表对象),其构造函数为:

```
csv.reader(csvfile, dialect = 'excel', ** fmtparams)          # 构造函数
```

其中,csvfile 是文件对象或 list 对象;dialect 用于指定 CSV 的格式模式,不同程序输出的 CSV 格式有细微差别;fmtparams 用于指定特定格式,以覆盖 dialect 中的格式。

csv.reader 对象是可迭代对象。reader 对象包含以下属性:

- csvreader.dialect:返回其 dialect。
- csvreader.line_num:返回读入的行数。

【例 16.9】　使用 csv.reader 对象读取 CSV 文件(csv_reader1.py)。

```
import csv
def readcsv1(csvfilepath):
    with open(csvfilepath, newline = '') as f:              # 打开文件
        f_csv = csv.reader(f)                               # 创建 csv.reader 对象
        headers = next(f_csv)                               # 标题
        print(headers)                                      # 打印标题(列表)
        for row in f_csv:                                   # 循环打印各行(列表)
            print(row)
if __name__ == '__main__':
    readcsv1(r'scores.csv')
```

程序运行结果如下。

```
['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
['101511', '宋颐园', '男', '一班', '72', '85', '82']
['101513', '王二丫', '女', '一班', '75', '82', '51']
['101531', '董再永', '男', '三班', '55', '74', '79']
['101521', '陈香燕', '女', '二班', '80', '86', '68']
['101535', '周一萍', '女', '三班', '72', '76', '72']
```

16.7.2　csv.writer 对象和 CSV 文件的写入

csv.writer 对象用于把列表对象数据写入到 CSV 文件,其构造函数为:

```
csv.writer(csvfile, dialect = 'excel', ** fmtparams)          # 构造函数
```

其中,csvfile 是任何支持 write()方法的对象,通常为文件对象;dialect 和 fmtparams 与 csv.reader 对象构造函数中参数的意义相同。

csv.writer 对象支持下列方法和属性:

- csvwriter.writerow(row):方法,写入一行数据。
- csvwriter.writerows(rows):方法,写入多行数据。
- csvreader.dialect:只读属性,返回其 dialect。

【例 16.10】　使用 csv.writer 对象写入 CSV 文件(csv_writer1.py)。

```
import csv
```

```
def writecsv1(csvfilepath):
    headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
    rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
            ('101513', '王二丫', '女', '一班', '75', '82', '51')]
    with open(csvfilepath,'w', newline='') as f:          #打开文件
        f_csv = csv.writer(f)                              #创建 csv.writer 对象
        f_csv.writerow(headers)                            #写入一行(标题)
        f_csv.writerows(rows)                              #写入多行(数据)
if __name__ == '__main__':
    writecsv1(r'scores1.csv')
```

16.7.3 csv.DictReader 对象和 CSV 文件的读取

使用 csv.reader 对象从 CSV 文件读取数据,结果为列表对象 row,需要通过索引 row[i] 访问。如果希望通过 CSV 文件的首行标题字段名访问,则可以使用 csv.DictReader 对象的构造函数,以返回 map:

```
csv.DictReader(csvfile, fieldnames=None, restkey=None, restval=None, dialect='excel',
 *args, **kwds)
```

其中,csvfile 是文件对象或 list 对象;fieldnames 用于指定字段名,如果没有指定,则第一行为字段名;可选的 restkey 和 restval 用于指定字段名和数据个数不一致时所对应的字段名或数据值;其他参数同 csv.reader 对象。

除了支持 csv.reader 对象的方法和属性以外,csv.DictReader 还支持下列属性:

```
csvreader.fieldnames                                       #返回标题字段名
```

【例 16.11】 使用 csv.DictReader 对象读取 CSV 文件(csv_reader2.py)。

```
import csv
def readcsv2(csvfilepath):
    with open(csvfilepath, newline='') as f:               #打开文件
        f_csv = csv.reader(f)                              #创建 csv.DictReader 对象
        headers = next(f_csv)                              #标题
        print(headers)                                     #打印标题(列表)
        for row in f_csv:                                  #循环打印各行(列表)
            print(row)
if __name__ == '__main__':
    readcsv2(r'scores.csv')
```

16.7.4 csv.DictWriter 对象和 CSV 文件的写入

如果需要写入 map 数据到 CSV,可以使用 csv.DictWriter 对象的构造函数:

```
csv.DictWriter(csvfile, fieldnames, restval='', extrasaction='raise', dialect='excel',
 *args, **kwds)
```

其中,csvfile 是文件对象或 list 对象;fieldnames 用于指定字段名;可选的 restval 用于指定默认数据;可选的 extrasaction 用于指定多余字段时的操作;其他参数同 csv.writer 对象。

除了支持 csv.writer 对象的方法和属性以外,csv.DictWriter 还支持如下方法:

```
DictWriter.writeheader()                                   #写入标题字段名(构造函数中的参数)
```

【例 16.12】 使用 csv.DictWriter 对象写入 CSV 文件(csv_writer2.py)。

```
import csv
```

```
def writecsv2(csvfilepath):
    headers = ['学号', '姓名', '语文', '数学', '英语']
    rows = [{'学号': '101511', '姓名': '宋颐园', '语文': '72', '数学': '85', '英语': '82'},
            {'学号': '101513', '姓名': '王二丫', '语文': '75', '数学': '82', '英语': '51'}]
    with open(csvfilepath,'w', newline = '') as f:          #打开文件
        f_csv = csv.DictWriter(f, headers)                  #创建 csv.DictWriter 对象
        f_csv.writeheader()                                 #写入标题
        f_csv.writerows(rows)                               #写入多行(数据)
if __name__ == '__main__':
    writecsv2(r'scores2.csv')
```

16.7.5　CSV 文件格式化参数和 Dialect 对象

1. CSV 文件格式化参数

在创建 reader/writer 对象时可以指定 CSV 文件格式化参数,CSV 文件格式化参数包括如下选项。

- delimiter:项目分隔符,默认为'.'。
- doublequote:如果为 True(默认值),字符串中的双引号使用""表示;如果为 False,使用转义字符 escapechar 指定的字符。
- escapechar:转义字符,默认为 None。
- lineterminator:用于 writer 的换行符,默认为'\r\n'。
- quotechar:用于限定包含特殊字符的字段的符号,默认为"。
- quoting:用于指定使用双引号的规则,可以为 csv 模块中的常量 QUOTE_ALL(全部)、QUOTE_MINIMAL(仅特殊字符字段)、QUOTE_NONNUMERIC(非数字字段)、QUOTE_NONE(全部不)。
- skipinitialspace:如果为 True,省略分隔符前面的空格;默认值为 False。
- strict:如果为 True,读入错误格式 CSV 行时将导致 csv.Error;默认值为 False。

【例 16.13】 CSV 文件格式化参数示例(csv_writer3.py)。

```
import csv
def writecsv3(csvfilepath):
    headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
    rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
            ('101513', '王二丫', '女', '一班', '75', '82', '51')]
    with open(csvfilepath,'w', newline = '') as f:
        f_csv = csv.writer(f, delimiter = ':', quoting = csv.QUOTE_ALL)
                                                            #指定格式化参数
        f_csv.writerow(headers)                             #写入一行(标题)
        f_csv.writerows(rows)                               #写入多行(数据)
if __name__ == '__main__':
    writecsv3(r'scores3.csv')
```

2. Dialect 对象

若干格式化参数可以组成 Dialect 对象,Dialect 对象包含对应于命名格式化参数的属性。用户可以创建 Dialect 或其派生类的对象,然后传递给 reader 或 writer 的构造函数。

可以使用下列 csv 模块的函数创建 Dialect 对象。

- csv.register_dialect(name[, dialect], ** fmtparams):使用命名参数,注册一个名称。

- csv.unregister_dialect(name)：取消注册的名称。
- csv.get_dialect(name)：获取注册名称的 Dialect 对象，无注册时将导致 csv.Error。
- csv.list_dialects()：所有注册对象 Dialect 的列表。

例如：

```
>>> import csv
>>> csv.list_dialects()                    #输出:['excel', 'excel-tab', 'unix']
```

另外，可以使用 csv 模块的 field_size_limit()函数获取和设置字段长度限制，其函数形式如下：

```
csv.field_size_limit([new_limit])
```

【例 16.14】 Dialect 对象示例(csv_writer4.py)。

```
import csv
def writecsv4(csvfilepath):
    csv.register_dialect('mydialect', delimiter = ':', quoting = csv.QUOTE_NONE)
    headers = ['学号', '姓名', '性别', '班级', '语文', '数学', '英语']
    rows = [('101511', '宋颐园', '男', '一班', '72', '85', '82'),
            ('101513', '王二丫', '女', '一班', '75', '82', '51')]
    with open(csvfilepath, 'w', newline = '') as f:
        f_csv = csv.writer(f, 'mydialect')  #指定格式化参数
        f_csv.writerow(headers)             #写入一行(标题)
        f_csv.writerows(rows)               #写入多行(数据)
if __name__ == '__main__':
    writecsv4(r'scores4.csv')
```

16.8 输入重定向和管道

fileinput 模块提供了用于循环处理输入、输入重定向、管道或一个或者多个文本文件的函数和辅助对象。

16.8.1 FileInput 对象

fileinput 模块的 FileInput 对象用于循环处理多个文件、重定向输入或管道。该对象用于循环遍历，支持上下文管理协议，其构造函数为：

```
fileinput.FileInput(files = None, inplace = False, backup = '', bufsize = 0, mode = 'r', openhook = None)
```

其中，files 是文件路径的元组；mode 是文件打开模式，默认为文本读取模式。例如：

```
with FileInput(files = ('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

使用 for…in 可以循环遍历文件列表中各文件的行。当前读取的文件、行等全局信息可以使用 FileInput 对象 f 的方法获取，具体形式如下。

- f.filename()：返回当前读取文件的文件名，读取第 1 个文件前为 None。
- f.fileno()：返回当前读取文件的文件名，无法打开文件时为－1。
- f.lineno()：返回累计读取的行数。
- f.filelineno()：返回当前文件读取的行数。
- f.isfirstline()：判断是否为当前读取文件的首行。
- f.isstdin()：判断最后读入的行是否为从标准输入 stdin 读入。

- f.nextfile()：关闭当前读取文件，读取下一文件。
- f.close()：关闭所有文件。

注意：fileinput 模块中包含与上述 FileInput 对象 f 的方法同名的模块函数，实现相同的功能。

16.8.2 fileinput 模块的函数

通常，使用 fileinput 模块的 input 函数可以返回 FileInput 对象，其函数形式如下：

```
fileinput.input(files = None, inplace = False, backup = '', bufsize = 0, mode = 'r', openhook = None)
```

其中，files 是文件路径的元组。例如：

```
with fileinput.input(files = ('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

如果输入的文件为压缩文件，或者是非 utf8 编码文件，则可以使用 fileinput 模块的 hook_compressed()函数或 hook_encoded()传递给 FileInput 的构造函数参数 openhook，具体函数形式如下。

- fileinput.hook_compressed(filename, mode)：openhook 函数，用于压缩文件。
- fileinput.hook_encoded(encoding)：openhook 函数，用于编码文件。

压缩文件支持'.gz'和'.bz2'，根据扩展名自动使用模块 gzip 或 bz2。例如：

```
fi1 = fileinput.FileInput(openhook = fileinput.hook_compressed)
fi2 = fileinput.FileInput(openhook = fileinput.hook_encoded("iso - 8859 - 1"))
```

【例 16.15】 fileinput 示例(fileinput_1.py)。

```
import fileinput, glob
def main():
    txtfiles = glob.glob(r'c:\pythonpa\ * .txt')
    with fileinput.input(files = txtfiles) as f:
        for line in f:
            print(f.filename(), f.lineno(), line, end = '')
if __name__ == '__main__':
    main()
```

程序运行结果如下。

```
c:\pythonpa\data1.txt 1 123
c:\pythonpa\data1.txt 2 abc
c:\pythonpa\data1.txt 3 456
c:\pythonpa\data1.txt 4 def
```

16.8.3 输入重定向

UNIX/Linux 和 Windows 均支持输入重定向。

【例 16.16】 使用 fileinput 实现输入重定向(fileinput_2.py)。

```
import fileinput
def main():
    with fileinput.input() as f:
        for line in f:
            print(f.filename(), f.lineno(), line, end = '')
if __name__ == '__main__':
    main()
```

1. 从文件输入

从文件输入的执行过程和运行结果如下。

```
C:\Pythonpa\ch16>fileinput_2.py fileinput_1.py fileinput_2.py
fileinput_1.py 1 import fileinput, glob
fileinput_1.py 2 def main():
fileinput_1.py 3     txtfiles = glob.glob(r'c:\pythonpa\*.txt')
fileinput_1.py 4     with fileinput.input(files=txtfiles) as f:
fileinput_1.py 5         for line in f:
fileinput_1.py 6             print(f.filename(), f.lineno(), line, end='')
fileinput_1.py 7 if __name__ == '__main__':
fileinput_1.py 8     main()
fileinput_2.py 9 import fileinput
fileinput_2.py 10 def main():
fileinput_2.py 11     with fileinput.input() as f:
fileinput_2.py 12         for line in f:
fileinput_2.py 13             print(f.filename(), f.lineno(), line, end='')
fileinput_2.py 14 if __name__ == '__main__':
fileinput_2.py 15     main()
```

2. 从管道输入

从管道输入的执行过程和运行结果如下。

```
C:\Pythonpa\ch16>dir | fileinput_2.py
<stdin> 1  驱动器 C 中的卷是 Windows7
<stdin> 2  卷的序列号是 1234-5678
<stdin> 3
<stdin> 4  C:\pythonpa\ch16 的目录
<stdin> 5
<stdin> 6 2018/07/10  13:33    <DIR>          .
<stdin> 7 2018/07/10  13:33    <DIR>          ..
<stdin> 8 2018/07/10  14:47                81 binaryread.py
…
```

3. 输入重定向

输入重定向的执行过程和运行结果如下。

```
C:\Pythonpa\ch16>fileinput_2.py < fileinput_2.py
<stdin> 1 import fileinput
<stdin> 2 def main():
<stdin> 3     with fileinput.input() as f:
<stdin> 4         for line in f:
<stdin> 5             print(f.filename(), f.lineno(), line, end='')
<stdin> 6 if __name__ == '__main__':
<stdin> 7     main()
```

16.9 对象序列化

16.9.1 对象序列化概述

在程序运行时,其数据对象创建在内存中。如果需要将其持久保存到磁盘,或通过网络传递给其他计算机,则需要通过对象序列化机制。

对象序列化也称为串行化,即将对象转换为数据形式,并转储到磁盘文件或通过网络实现跨平台传输。反过来,从磁盘数据文件或接收到的数据形式恢复,以得到相应对象的过程,则

称为反序列化。

多个对象可以串行化转储到一个磁盘文件，用户不必关心数据的格式。对象序列化机制被广泛用于各种分布式并行处理系统。

使用 pickle/cPickle 模块中提供的函数可以实现 Python 对象的序列化。cPickle 使用 C 语言实现，故其运行效率更高。另外，marshal 模块也提供了类似的函数，但 pickle 模块更具有普遍意义。

16.9.2　pickle 模块和对象序列化

pickle 模块实现了 Python 数据对象的序列化和反序列化。

```
pickle.dump(obj, file, protocol = None)        #将对象 obj 保存到文件 file 中
pickle.load(file)                              #从 file 中读取并重构一个对象
```

其中，protocol 为序列化使用的协议版本，可取值为 0（默认值，ASCII 协议，所序列化的对象使用可打印的 ASCII 码表示）、1（二进制协议）、2（二进制协议，2.3 版本）和 3（二进制协议，3.0 版本）。

【例 16.17】　对象序列化示例（pickledump.py）。

```
import pickle
with open(r'c:\pythonpa\dataObj1.dat', 'wb') as f:
    s1 = 'Hello!'
    c1 = 1 + 2j
    t1 = (1,2,3)
    d1 = dict(name = 'Mary', age = 19)
    pickle.dump(s1, f)
    pickle.dump(c1, f)
    pickle.dump(t1, f)
    pickle.dump(d1, f)
```

【例 16.18】　对象反序列化示例（pickleload.py）。

```
import pickle
with open(r'c:\pythonpa\dataObj1.dat', 'rb') as f:
    o1 = pickle.load(f)
    o2 = pickle.load(f)
    o3 = pickle.load(f)
    o4 = pickle.load(f)
    print(type(o1), str(o1))
    print(type(o2), str(o2))
    print(type(o3), str(o3))
    print(type(o4), str(o4))
```

程序运行结果如下。

```
<class 'str'> Hello!
<class 'complex'> (1 + 2j)
<class 'tuple'> (1, 2, 3)
<class 'dict'> {'age': 19, 'name': 'Mary'}
```

16.9.3　json 模块和 JSON 格式数据

json 模块类似于 pickle 模块，也可以实现对象序列化，并可与其他语言编写的程序实现数据交换。

JSON(JavaScript Object Notation,JavaScript 对象标记)定义了一种标准格式,用字符串来描述典型的内置对象(例如字典、列表、数字和字符串)。虽然 JSON 原来是 JavaScript 编程语言的一个子集,但它现在是一个独立于语言的数据格式,所有主流编程语言都有生产和消费的 JSON 数据的库。JSON 是网络数据交换的流行格式之一。

Python 标准库模块 json 包含将 Python 对象编码为 JSON 格式(或者简称 JSON)和将 JSON 解码到 Python 对象的函数。

- dumps(obj):把 obj 对象序列化为 JSON 字符串。
- dump(obj, fp):把 obj 对象序列化为 JSON 字符串写入文件 fp。
- loads(s):返回把 JSON 字符串 s 反序列化后的对象。
- load(fp):返回把从文件 fp 中读取的 JSON 字符串反序列化后的对象。

例如:

```
>>> import json
>>> data = [{'a': 'A', 'b': (2, 4), 'c': 3.0}]
>>> str_json = json.dumps(data)
>>> str_json                          #输出:'[{"a": "A", "c": 3.0, "b": [2, 4]}]'
>>> data1 = json.loads(str_json)
>>> data1                             #输出:[{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
```

【例 16.19】 对象 JSON 格式序列化示例(json_dump.py)。

```
import json
urls = {'baidu':'http://www.baidu.com/',
       'sina':'http://www.sina.com.cn/',
       'tencent':'http://www.qq.com/',
       'taobao':'https://www.taobao.com/'}
with open(r'c:\pythonpa\data.json', 'w') as f:
    json.dump(urls, f)
```

【例 16.20】 对象 JSON 格式反序列化示例(json_load.py)。

```
import json
with open(r'c:\pythonpa\data.json', 'r') as f:
    urls = json.load(f)
    print(urls)
```

程序运行结果如下。

```
{'tencent': 'http://www.qq.com/', 'sina': 'http://www.sina.com.cn/', 'taobao': 'https://www.taobao.
com/', 'baidu': 'http://www.baidu.com/'}
```

16.10 复 习 题

一、填空题

1. Python 可以使用函数________打开文件。

2. 文件操作可以使用________方法关闭流,以释放资源。通常采用________语句,以保证系统自动关闭打开的流。

3. 在打开随机文件后,可以使用实例方法________进行定位。

4. ________模块提供了用于循环处理输入、输入重定向、管道或一个或多个文本文件的函数和辅助对象。

5. 可以使用________模块中提供的函数实现 Python 对象的序列化。

二、思考题

1. 在使用 open()函数时,指定打开文件的模式 mode 有哪几种? 其默认打开模式是什么?

2. 文本文件的读取和写入的基本步骤是什么?

3. 二进制文件的打开模式是什么? 简述其读取和写入的基本步骤。

4. 如何随机访问文件? 其打开模式是什么?

5. Python 如何实现内存文本文件和内存二进制文件的读取操作?

6. Python 如何实现压缩文件的读取和写入操作?

7. 如何实现 Python 对象的序列化和反序列化?

8. 如何使用 os 模块提供的函数读取和写入文件?

16.11 上机实践

完成本章中的例 16.1~例 16.20,熟悉 Python 语言中的文件和数据交换程序设计。

16.12 案例研究:百度音乐批量下载器

本章案例研究通过图形用户界面的百度音乐批量下载器的实现,帮助读者深入了解在图形用户界面程序中使用文件和数据交换的思维和流程。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

第 17 章

数据库访问



视频讲解

应用程序往往使用数据库来存储大量的数据。Python 提供了对大多数数据库的支持。使用 Python 中相应的模块可以连接到数据库,进行查询、插入、更新和删除等操作。

17.1 数据库基础

17.1.1 数据库的概念

数据库就是存储数据的仓库,即存储在计算机系统中结构化的、可共享的相关数据的集合。数据库中的数据按一定的数据模型组织、描述和存储,可以最大限度地减少数据的冗余度。

数据库管理系统(Database Management System,DBMS)是用于管理数据的计算机软件。数据库管理系统使用户能够方便地定义数据、操作数据以及维护数据。其主要功能如下。

(1) 数据定义功能:使用数据定义语言(Data Definition Language,DDL)可以生成和维护各种数据对象的定义。

(2) 数据操作功能:使用数据操作语言(Data Manipulation Language,DML)可以对数据库进行查询、插入、删除和修改等基本操作。

(3) 数据库的管理和维护:数据库的安全性、完整性、并发性、备份和恢复等功能。

目前流行的数据库管理系统产品可以分为以下两类。

(1) 适合于企业用户的网络版 DBMS,例如 Oracle、Microsoft SQL Server、IBM DB2 等。

(2) 适合于个人用户的桌面 DBMS,例如 Microsoft Access 等。

数据库系统(Database System,DBS)是指在计算机系统中引入数据库后组成的系统。数据库系统一般包括计算机硬件、操作系统、DBMS、开发工具、应用系统、数据库管理员(Database Administrator,DBA)和用户等。

17.1.2 关系数据库

常用的数据库模型包括层次模型(Hierarchical Model)、网状模型(Network Model)、关系模型(Relational Model)和面向对象的数据模型(Object Oriented Model)。

关系模型具有完备的数学基础,简单灵活,易学易用,已经成为数据库的标准。目前流行的 DBMS 都是基于关系模型的关系数据库管理系统。

关系模型把世界看作是由实体(Entity)和联系(Relationship)构成的。实体是指现实世界中具有一定特征或属性并与其他实体有联系的对象,在关系模型中实体通常是以表的形式

来表现。表的一行描述实体的一个实例，表的一列描述实体的一个特征或属性。

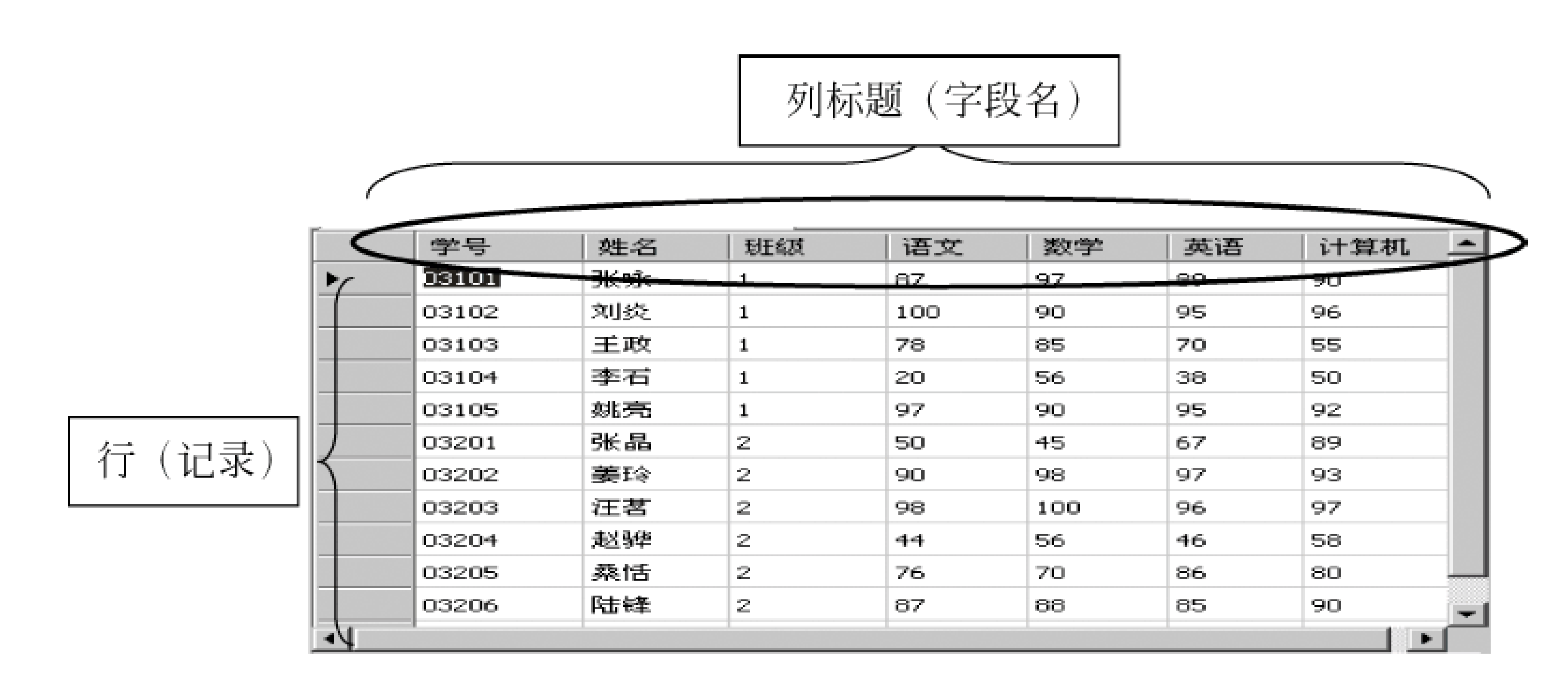
联系是指实体之间的对应关系，通过联系就可以用一个实体的信息来查找另一个实体的信息。联系可以分为以下 3 种。

- 一对一：例如一个部门只能有一个经理，而一个经理只能在一个部门任职，部门和经理为一对一的联系。
- 一对多：例如一个部门有多名员工，而一名员工只能在一个部门工作，部门和员工为一对多的联系。
- 多对多：例如一名学生可以选修多门课程，而一门课程可以有多名选修的学生，学生和课程是多对多的联系。

在关系数据库中，常见的数据库对象包括表、视图、触发器、存储过程等。

数据库中的表由行(Row)和列(Column)组成。列由同类的信息组成，又称为字段(Field)；列的标题称为字段名。行是指包括了若干列信息项的一行数据，也称为记录(Record)或元组(Tuple)。一个数据库表由一条或多条记录组成，没有记录的表称为空表。

每个数据表中通常都有一个主关键字(Primary Key)，用于唯一确定一条记录，例如图 17-1 中的“学号”字段即为 Exam 数据表的主关键字。



学号	姓名	班级	语文	数学	英语	计算机
03101	张咏	1	87	97	89	90
03102	刘炎	1	100	90	95	96
03103	王政	1	78	85	70	55
03104	李石	1	20	56	38	50
03105	姚亮	1	97	90	95	92
03201	张晶	2	50	45	67	89
03202	姜玲	2	90	98	97	93
03203	汪茗	2	98	100	96	97
03204	赵骅	2	44	56	46	58
03205	桑恬	2	76	70	86	80
03206	陆锋	2	87	88	85	90

图 17-1　数据表的行(记录)和列(字段)信息

17.2　Python 数据库访问模块

17.2.1　通用数据库访问模块

1. ODBC

ODBC(Open Database Connectivity，开放数据库互连)提供了一种标准的应用程序编程接口(Application Programming Interface，API)方法来访问数据库管理系统。

在 Windows 平台上，常用的数据库产品都实现了其各自的 ODBC 驱动程序，包括 Oracle、DB2、Microsoft SQL Server、Access 等数据库。通过 ODBC 可以实现通用的数据库访问，Python 提供了如下几种模块以访问 ODBC。

- ODBC Interface：随 PythonWin 附带发行的模块。
- pyodbc：开源的 Python ODBC 接口，完整实现了 DB-API 2.0 接口。
- mxODBC：流行的 mx 系列工具包中的一部分(非商业开发需付费)，实现了绝大部分 DB-API 2.0 接口。

2. JDBC

JDBC(Java Database Connectivity,Java 数据库连接)是基于 Java 的面向对象的应用编程接口,描述了一套访问关系数据库的 Java 类库标准。

在 Python 2.1 以后的发行版中,包括通过 JDBC 访问数据的模块 zxJDBC,建立在底层的 JDBC 接口之上,支持 DB-API 2.0 接口。

17.2.2 专用数据库访问模块

Python 针对各种流行的数据库,提供了各种专用的数据库访问模块,参见表 17-1。

表 17-1 Python 专用数据库访问模块

数 据 库	Python 模块	网 址
MySQL	mysql-python	http://sourceforge.net/projects/mysql-python
PostgreSQL	PyGreSQL	http://www.pygresql.org/
Oracle	DCOracle2	http://www.zope.org/Members/matt/dco2
IBM DB2	pydb2	http://sourceforge.net/projects/pydb2
SQL Server	pymssql	http://pymssql.sourceforge.net/

17.2.3 SQLite 数据库和 sqlite3 模块

1. SQLite 数据库

SQLite 是一款开源的轻型数据库,占用的资源非常低,广泛用于各种嵌入式设备中。SQLite 支持各种主流的操作系统,包括 Windows、Linux、UNIX 等,并与许多程序语言紧密结合,包括 Python。

SQLite 是遵守 ACID(原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability))的关系数据库管理系统,实现了多数的 SQL-92 标准,包括事务、触发器和多数的复杂查询。

SQLite 不进行类型检查,例如可以把字符串插入整数列中。该特点特别适于和无类型的脚本语言(例如 Python)一起使用。

SQLite 整个数据库包括数据库定义、表、索引和数据本身等,都存储在一个单一的文件中,其事务处理通过锁定整个数据文件完成。

SQLite 引擎不是程序与之通信的独立进程,而是在编程语言内直接调用 API 来实现,即 SQLite 是应用程序的组成部分,所以具有内存消耗低、延迟时间短、整体结构简单等优点。

SQLite 目前的版本是 3,其官方网址为 http://www.sqlite.org。

2. SQLite 支持的数据类型

SQLite 支持的数据类型包括 NULL、INTEGER、REAL、TEXT 和 BLOB,分别对应 Python 的数据类型 None、int、float、str 和 bytes。

用户可以使用适配器(adapters),以将更多的 Python 类型存储到 SQLite 数据库;也可以使用转换器(converters),把 SQLite 数据类型转换为 Python 的数据类型。

3. sqlite3 模块

Python 标准模块 sqlite3 使用 C 语言实现,提供了访问和操作数据库 sqlite 的各种功能。sqlite3 模块主要包括下列常量、函数和对象:

- sqlite3.version　　　　　　　　# 常量，版本号
- sqlite3.connect(database)　　　# 函数，连接到数据库，返回 Connect 对象
- sqlite3.Connect　　　　　　　　# 数据库连接对象
- sqlite3.Cursor　　　　　　　　 # 游标对象
- sqlite3.Row　　　　　　　　　　# 行对象

17.3　使用 sqlite3 模块连接和操作 SQLite 数据库

17.3.1　访问数据库的步骤

Python 的数据库模块具有统一的接口标准，数据库操作遵循一致的模式。使用 sqlite3 模块操作数据的步骤如下。

1. 导入相应的数据库模块

Python 标准库中带有 sqlite3 模块，可以使用如下命令直接导入：

```
import sqlite3
```

2. 建立数据库连接，返回 Connection 对象

使用数据库模块的 connect()函数建立数据库连接，返回连接对象 con，命令形式如下：

```
con = sqlite3.connect(connectstring)      # 连接到数据库，返回 sqlite3.Connection 对象
```

其中，connectstring 是连接字符串。对于不同的数据库连接对象，其连接字符串的格式各不相同。sqlite 的连接字符串为数据库的文件名，例如“c:\example.db”。如果指定连接字符串为“：memory：”，则可以创建一个内存数据库。例如：

```
>>> import sqlite3
>>> con = sqlite3.connect("c:\db1.db")
```

如果“c:\db1.db”存在，则打开数据库；否则创建并打开数据库“c:\db1.db”。

在创建数据库连接对象（Connection 对象）后可以设置其属性。例如：

```
>>> con.isolation_level = None          # 设置事务隔离级别，默认为自动提交
>>> con.row_factory = sqlite3.Row       # 设置连接对象使用的行工厂对象
```

3. 创建游标对象 cur

使用如下命令可以调用 con.cursor()函数创建游标对象 cur：

```
cur = con.cursor()                      # 创建游标对象
```

4. 使用 Cursor 对象的 execute 执行 SQL 命令返回结果

调用 Cursor 对象的 execute()/executemany()/executescript()方法查询数据库，其调用形式如下：

- cur.execute(sql)　　　　　　　　　　　# 执行 SQL 语句
- cur.execute(sql, parameters)　　　　　# 执行带参数的 SQL 语句
- cur.executemany(sql, seq_of_parameters)　# 根据参数执行多次 SQL 语句
- cur.executescript(sql_script)　　　　　# 执行 SQL 脚本

一般建议直接使用 Connection 对象的 execute()、executemany()、executescript()方法。事实上，它们是 Cursor 对象对应方法的快捷方式，系统创建一个临时 Cursor 对象，然后调用对应的方法，并返回 Cursor 对象。具体如下：

- con.execute(sql)　　　　　　　　　　　# 执行 SQL 语句，返回结果

```
• con.execute(sql, parameters)                  # 执行带参数的 SQL 语句，返回结果
• con.executemany(sql, seq_of_parameters)       # 根据参数执行多次 SQL 语句，返回结果
• con.executescript(sql_script)                 # 执行 SQL 脚本
```

例如：

```
>>> con.execute("create table if not exists t1(id primary key, name)")
```

此时将创建一个包含 id（主码）和 name 两个字段的表 t1。

在 SQL 语句字符串中可以使用占位符"?"表示参数，传递的参数使用元组；或者使用命名参数，传递的参数使用字典。例如：

```
>>> con.execute("insert into t1(id, name) values (?, ?)", ('001', '北京'))
>>> con.execute("insert into t1(id, name) values (:id, :name)", {'id':'027', 'name':'武汉'})
```

5. 获取游标的查询结果集

调用 cur.fetchall()、cur.fetchone()、cur.fetchmany()返回查询结果，调用形式如下：

```
• cur.fetchone()          # 返回结果集的下一行(Row 对象);无数据时返回 None
• cur.fetchall()          # 返回结果集的剩余行(Row 对象列表);无数据时返回空 list
• cur.fetchmany(size)     # 返回结果集的多行(Row 对象列表);无数据时返回空 list
• cur.rowcount            # 返回影响的行数、结果集的行数
```

Row 对象 r 为一行查询结果系列，支持下列访问：

```
• r[i]                    # 按索引访问，返回第 i 列的数据
• r[colname]              # 按列名称访问，返回 colname 列的数据
• len(r)                  # 返回列数
• tuple(r)                # 把数据转换为元组
• r.keys()                # 返回列名称的列表
```

例如：

```
>>> cur = con.cursor()                      # 创建游标对象
>>> cur.execute("select * from t1")         # 执行 SQL 查询
>>> r = cur.fetchone()                      # 获取一行 Row 对象结果
>>> r.keys()                                # 返回列名称的列表
>>> r[0], r['name']
```

当然也可以直接使用循环输出结果。例如：

```
>>> for row in con.execute("select * from t1"):
        print(row[0], row[1])
```

6. 数据库的提交和回滚

根据数据库事务隔离级别的不同，可以提交或回滚，命令形式如下：

```
• conn.commit()           # 提交
• conn.rollback()         # 回滚
```

7. 关闭 Cursor 对象和 Connection 对象

最后，需要关闭打开的 Cursor 对象和 Connection 对象，命令形式如下：

```
• cur.close()             # 关闭 Cursor 对象
• con.close()             # 关闭 Connection 对象
```

17.3.2 创建数据库和表

使用 sqlite3.connect("数据库文件名")可以创建或打开 SQLite 数据库，并返回连接对象 con；使用 con.execute("create table …")可以创建表。

【例 17.1】 创建数据库和表(DBCreate.py):创建数据库 sales,并在其中创建表 region,该表中包含 id 和 name 两个字段(列),其中 id 为主码(primary key)。

```
import sqlite3
#创建 SQLite 数据库:c:\Pythonpa\ch17\sales.db
con = sqlite3.connect(r"c:\Pythonpa\ch17\sales.db")
#创建表 region,包含两个列,即 id(主码)和 name
con.execute("create table region(id primary key, name)")
```

17.3.3 数据库表的插入、更新和删除操作

在数据库表中插入、更新和删除记录的一般步骤如下。

(1) 建立数据库连接。

(2) 根据 SQL Insert、Update、Delete 语句,使用 con.execute(sql)执行数据库记录的插入、更新、删除操作,并根据返回的值判断操作结果。

(3) 提交操作。

(4) 关闭数据库。

【例 17.2】 数据库表记录的插入、更新和删除操作示例(DBUpdate.py):在数据库 sales 的数据表 region 中插入、更新、删除若干条记录。

```
import sqlite3
regions = [("021", "上海"),('022', "天津"),("023", "重庆"),("024", "沈阳")]
#打开 SQLite 数据库:c:\Pythonpa\ch17\sales.db
con = sqlite3.connect(r"c:\Pythonpa\ch17\sales.db")
#使用不同的方法分别插入一行数据
con.execute("insert into region(id, name) values ('020', '广东')")
con.execute("insert into region(id, name) values (?, ?)", ('001', '北京'))
#插入多行数据
con.executemany("insert into region(id, name) values (?, ?)", regions)
#修改一行数据
con.execute("update region set name = ? where id = ?", ('广州', '020'))
#删除一行数据
n = con.execute("delete from region where id = ?", ("024",))
print('删除了', n.rowcount, '行记录')
con.commit()                                    #提交
con.close()                                     #关闭数据库
```

17.3.4 数据库表的查询操作

查询数据库表的一般步骤如下:

(1) 建立数据库连接。

(2) 根据 SQL Select 语句,使用 con.execute(sql)执行数据库查询操作,返回游标对象 cur。

(3) 循环输出结果。

【例 17.3】 查询数据表中的记录信息(DBquery.py):查询并输出数据库 sales 的数据表 region 中的所有记录内容。

```
import sqlite3
#打开 SQLite 数据库:c:\Pythonpa\ch17\sales.db
con = sqlite3.connect(r"c:\Pythonpa\ch17\sales.db")
#查询数据库表的记录内容
cur = con.execute("select id, name from region")
```

```
for row in cur:                          #循环输出结果
    print(row)
```

程序运行结果如下。

```
('020', '广州')
('001', '北京')
('021', '上海')
('022', '天津')
('023', '重庆')
```

17.4 使用 SQLiteStudio 查看和维护 SQLite 数据库

使用 SQLite 可视化工具可以方便地查看程序运行后更新数据库的结果。

【例 17.4】 下载、安装和使用 SQLiteStudio.exe。

(1) 下载 SQLiteStudio。在浏览器中输入 SQLiteStudio 官网地址"https://sqlitestudio.pl/",下载最新版本(sqlitestudio-3.1.1.zip)。

(2) 运行 SQLiteStudio。解压缩下载的文件到本地任何目录下,双击运行解压缩的程序 SQLiteStudio.exe,打开 SQLiteStudio。

(3) 打开数据库。按 Ctrl+O 组合键,选择打开数据"c:\pythonpa\cs\jwxt\jwxt.db",查看数据库中表的结果,或者查看表的数据,如图 17-2 所示。

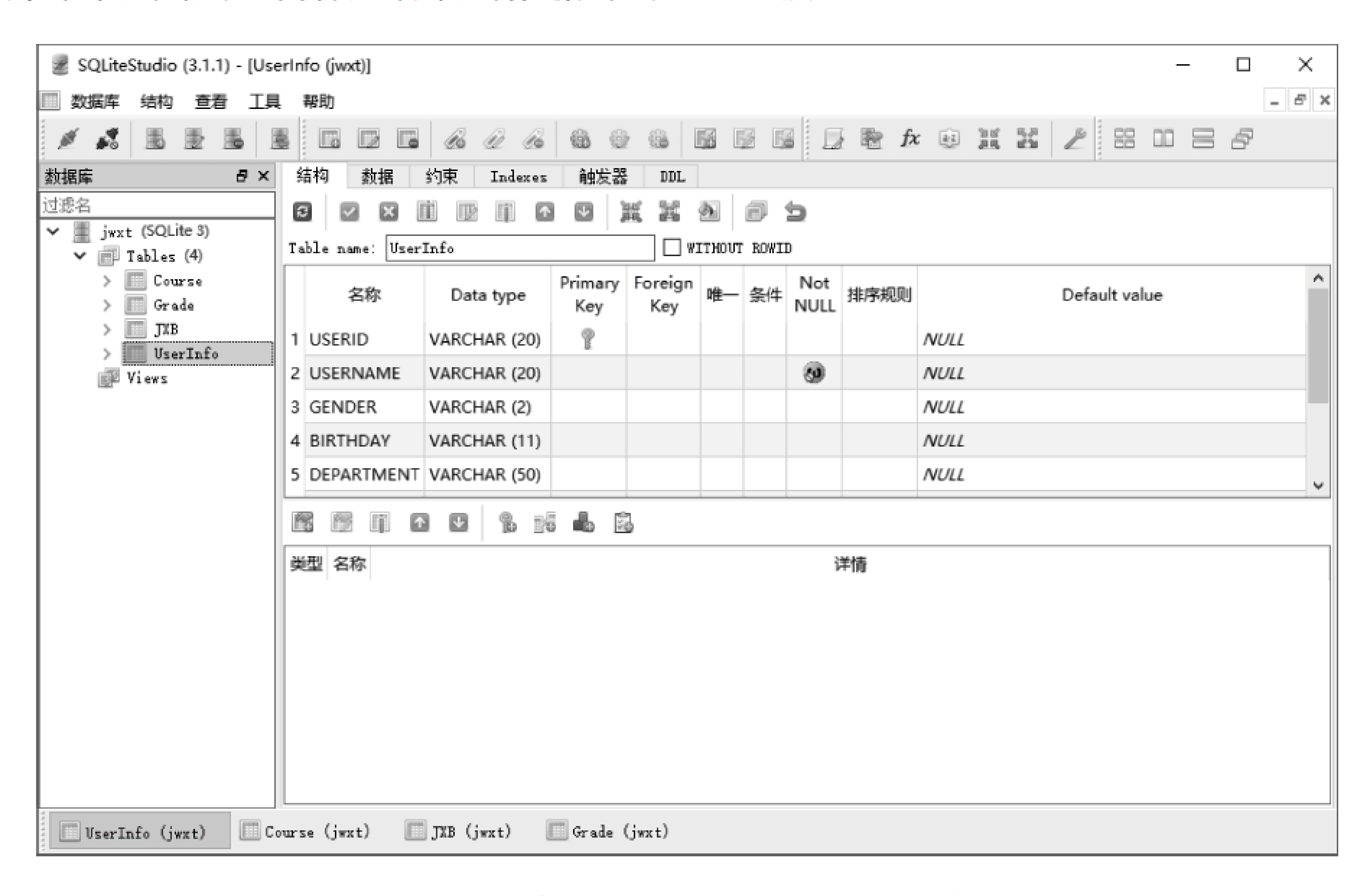


图 17-2 使用 SQLiteStudio 查看表数据

17.5 复 习 题

1. 数据库管理系统主要包括哪些功能?
2. 目前有哪些流行的数据库管理系统产品?
3. 常用的数据库模型是什么? 目前流行的 DBMS 主要基于哪类数据库模型?

4. Python 提供了哪几类数据库访问模块？
5. SQLite 支持哪几类数据类型？分别对应于 Python 的哪些数据类型？
6. sqlite3 模块主要包括哪些常量、函数和对象？
7. 使用 sqlite3 模块操作数据的典型步骤是什么？
8. Python 在数据库表中插入、更新和删除记录的一般步骤是什么？
9. Python 在数据库表中查询记录的一般步骤是什么？

17.6　上机实践

完成本章中的例 17.1～例 17.4，熟悉 Python 语言中的数据库访问程序设计。

17.7　案例研究：基于数据库和 GUI 的教务管理系统

本章案例研究通过图形用户界面的教务管理系统的设计与实现，帮助读者深入了解使用 wxPython 和数据库开发图形用户界面的数据库应用程序的思维和流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 18 章

网络编程和通信



视频讲解

Python 提供了用于网络编程和通信的各种模块，编程人员可以使用 socket 模块进行基于套接字的底层网络编程，也可以使用 urllib、http、ftplib、poplib、smtplib 等模块针对特定网络协议编程，还可以使用扩展库进行网络编程。

18.1 网络编程的基本概念

18.1.1 网络基础知识

计算机网络由通过传输介质连接在一起的一系列设备(网络节点)组成。一个节点可以是一台计算机、打印机或者是任何能够发送或接收由网络上其他节点产生数据的设备。

两台计算机之间要进行通信，必须采用相同的信息交换规则。在计算机网络中，用于规定信息的格式以及如何发送和接收信息的一套规则、标准或约定称为网络协议(Network Protocol)。目前使用最广泛的网络协议是 Internet 上所使用的 TCP/IP(Transmission Control Protocol/Internet Protocol)协议。

网络编程就是通过网络协议与其他计算机进行通信。网络编程涉及主机定位和数据传输。在 TCP/IP 协议中，TCP 层提供面向应用的数据传输机制；IP 层则负责网络主机定位、数据传输路由。

目前较为流行的网络编程模型是客户机/服务器(C/S)结构，如图 18-1 所示。

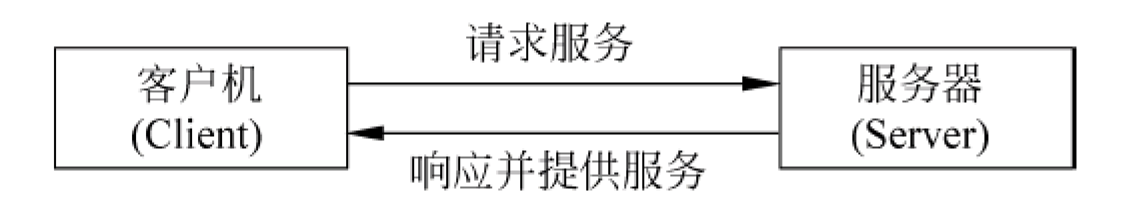


图 18-1 客户机/服务器(C/S)结构

事实上，C/S 模型体现的是一种网络数据访问的实现方式，请求服务的一方为客户机，响应请求并提供服务的一方为服务器，故一台主机有可能同时作为客户机角色和服务器角色。

18.1.2 TCP/IP 协议简介

TCP/IP 协议即传输控制协议/互联网协议，它是一种网际互联通信协议，目的在于通过它实现网际间各种异构网络和异种计算机的互联通信。众多的网络产品厂家都支持 TCP/IP 协议，TCP/IP 已经成为一个事实上的工业标准。TCP/IP 协议模型把 TCP/IP 协议族分成 4 个层次，如图 18-2 所示。

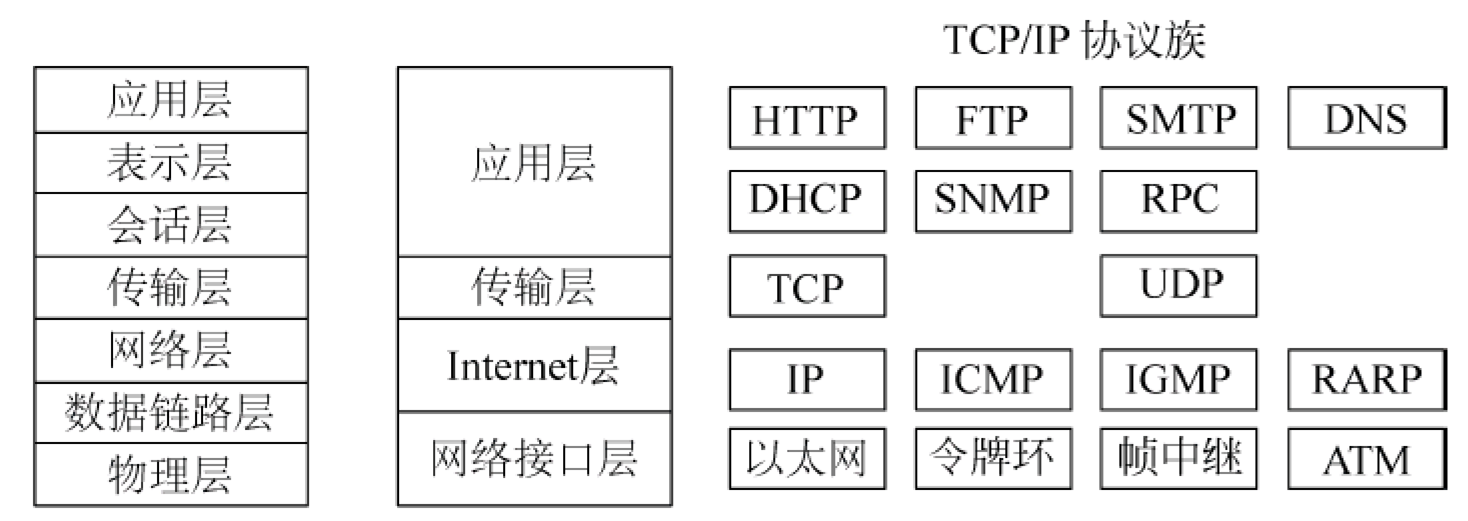


图 18-2 TCP/IP 四层参考模型

1. 网络接口层

网络接口层(又称网络访问层)对应于 OSI 模型的数据链路层和物理层,负责向网络媒体发送 TCP/IP 数据包并从网络媒体接收 TCP/IP 数据包。从理论上讲,该层不是 TCP/IP 协议的组成部分,但它是 TCP/IP 协议的基础,是各种网络与 TCP/IP 协议的接口。

2. Internet 层

Internet 层对应于 OSI 模型的网络层,负责相同或不同网络中计算机之间的通信,主要处理数据报和路由。IP 是一个可路由的协议,可以为数据包进行寻址、路由、分段和重组。IP 协议在 TCP/IP 协议组中处于核心地位。

3. 传输层

传输层对应于 OSI 模型的传输层,为应用层提供端到端的会话和数据报通信服务,主要功能是数据格式化、数据确认和丢失重传等。该层主要包括两种协议,即 TCP 协议和 UDP 协议。

1) TCP 协议

TCP 协议定义了两台计算机之间进行可靠传输时交换的数据和确认信息的格式,以及计算机为了确保数据的正确到达而采取的措施。该协议是面向连接的,可提供可靠的、按序传送数据的服务。TCP 协议采用的最基本的可靠性技术包括 3 个方面,即确认与超时重传、流量控制和拥塞控制。

TCP 协议的优点是可靠通信服务,缺点是建立连接需要额外的网络开销。

2) UDP 协议

UDP 协议(User Datagram Protocol,用户数据报协议)也是建立在 IP 协议之上,和 IP 协议一样提供无连接数据报传输。UDP 本身并不提供可靠性服务,相对 IP 协议,它唯一增加的功能是提供了协议端口,以保证进程通信。与 TCP 不同,UDP 提供一对一或一对多的、无连接的不可靠通信服务。

UDP 协议的优点是简单、高效,缺点是通信服务有可能不可靠。

4. 应用层

应用层是 TCP/IP 协议的最高层,对应于 OSI 模型的应用层、表示层和会话层。应用层允许应用程序访问其他层的服务,它定义了应用程序用来交换数据的协议。应用层中包含大量的协议,而且随着网络技术和应用的发展会不断产生许多新的应用层协议。

18.1.3 IP 地址和域名

1. IP 地址

在 Internet 中,网络中的两台主机进行通信时,其传送的数据包里必须包含附加信息的地址信息(即发送数据的计算机的地址和接收数据的计算机的地址),以保证通信主机间的正确路由。

Internet 采用一种全局通用的地址格式,为网络中的每一台主机都分配一个唯一的地址,称为 IP 地址。

Internet 中使用的 IPv4 版本的 TCP/IP 协议标准,规定 IP 地址由 32 位二进制数码组成。IP 地址在计算机中一般采用 32 位二进制位表示,例如 IP 地址 10101100 00010000 00000000 00000001。为了人工阅读方便,一般采用点分十进制表示方法,即 32 位二进制数码组成的 IP 地址,每 8 位为一组,共分为 4 组,中间用"."隔开。例如,IP 地址 10101100 00010000 00000000 00000001 用点分十进制表示法可以记为 172.16.0.1。

以 127 开头的 IP 地址(例如 127.0.0.1)为本机回送地址(Loopback Address),主要用于网络软件测试以及本地机进程间通信,无论什么程序,一旦使用回送地址发送数据,协议软件立即返回之,不进行任何网络传输。

2. 域名系统

Internet 上计算机之间的 TCP/IP 通信是通过 IP 地址来进行的,Internet 上的计算机都应该有一个唯一的 IP 地址。但是 IP 地址是基于数字来标识的,例如 64.233.189.104,可记忆性差,十分不友好,所以人们使用比较友好的计算机域名,例如 www.google.com。

Internet 使用域名系统(Domain Name System,DNS)来管理计算机域名与 IP 地址的对应关系。用户先在域名系统中注册域名及与其对应的 IP 地址。

当需要使用域名进行通信时,DNS 客户机通过查询 DNS 服务器将此域名解析为相对应的 IP 地址信息,然后通过 IP 地址进行通信。

18.1.4 统一资源定位器

IP 地址用来标识 Internet 上的主机,而位于 Internet 主机上的资源(例如各种文档、图像等)则通过统一资源定位器来标识。

URL(Uniform Resource Locator,统一资源定位器)是专为标识 Internet 网上资源位置而设的一种编址方式。通过 URL 可以访问 Internet 上的各种网络资源,比如最常见的 WWW、FTP 站点。浏览器通过解析给定的 URL 可以在网络上查找相应的文件或其他资源。URL 一般由以下几个部分组成:

传输协议://主机 IP 地址(或域名地址)[:端口号]/资源所在路径和文件名

其中,传输协议是指访问该资源所使用的访问协议;主机 IP 地址(或域名地址)是指资源所在的 Internet 主机;端口号是指主机上提供资源的服务的 TCP/IP 端口,如 http 使用 WWW 服务(默认端口为 80),ftp 表示 FTP 服务(默认端口为 21);路径是指资源所在路径和文件名。例如:

```
http://www.baidu.com/
http://home.yahoo.com:80/index.html
http://www.gamelan.com:80/Gamelan/network.html#BOTTOM
http://user:passwd@www.google.com/pages/index.html?key1=data1&key2=data2#faq
```

注意:TCP/IP 系统中的端口号是一个 16 位的数字,它的范围是 0～65 535。

18.2 基于 socket 的网络编程

18.2.1 socket 概述

1. 套接字

套接字是网络中两个应用程序之间通信的端点。网络上的两个程序通过一个双向的通信连接实现数据的交换,这个双向链路的一端就是一个 socket。

基于 TCP/IP 通信协议的 socket 由一个 IP 地址和一个端口号唯一确定。

TCP/IP 协议的传输层包含两个传输协议，即面向连接的 TCP 和非面向连接的 UDP。TCP 广泛用于各种可靠的传输，例如 HTTP、FTP、SMTP 等都使用 TCP 传输协议；UDP 不保证可靠传输，但其传输更简单、高效，故适合于实时交互性应用，例如音频、视频会议等。TCP 和 UDP 的程序架构各不相同。UDP 使用数据报传输数据。

2. TCP 通信程序设计

基于 socket 的面向连接的 TCP 网络程序的 C/S 架构如图 18-3 所示。

基于套接字的 TCP Server 的网络编程一般包括以下基本步骤。

(1) 创建 socket 对象。

(2) 将 socket 绑定到指定地址上。

(3) 准备好套接字，以便接收连接请求。

(4) 通过 socket 对象方法 accept()等待客户请求连接。

(5) 服务器和客户机通过 send()和 recv()方法通信（传输数据）。

(6) 传输结束，调用 socket 的 close()方法关闭连接。

其中，第(5)步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

基于套接字的 TCP Client 的网络编程一般包括以下基本步骤。

(1) 创建 socket 对象。

(2) 通过 socket 对象方法 connect()连接服务器。

(3) 客户机和服务器通过 send()和 recv()方法通信（传输数据）。

(4) 传输结束，调用 socket 的 close()方法关闭连接。

其中，第(3)步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

3. UDP 通信程序设计

基于 socket 的非面向连接的 UDP 网络程序的 C/S 架构如图 18-4 所示。

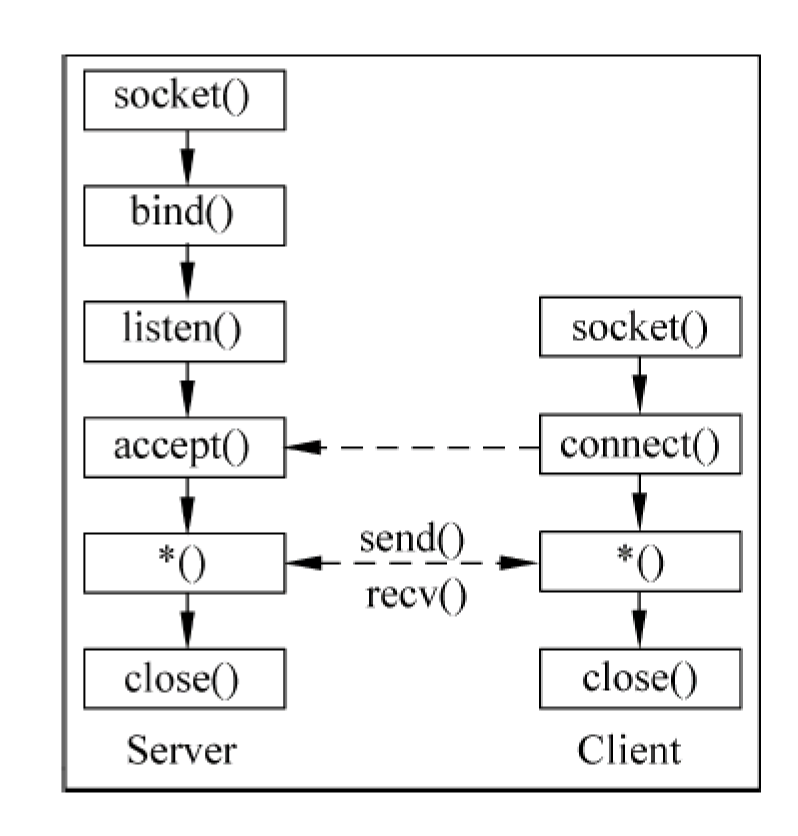




图 18-3　基于 socket 的 TCP 程序架构

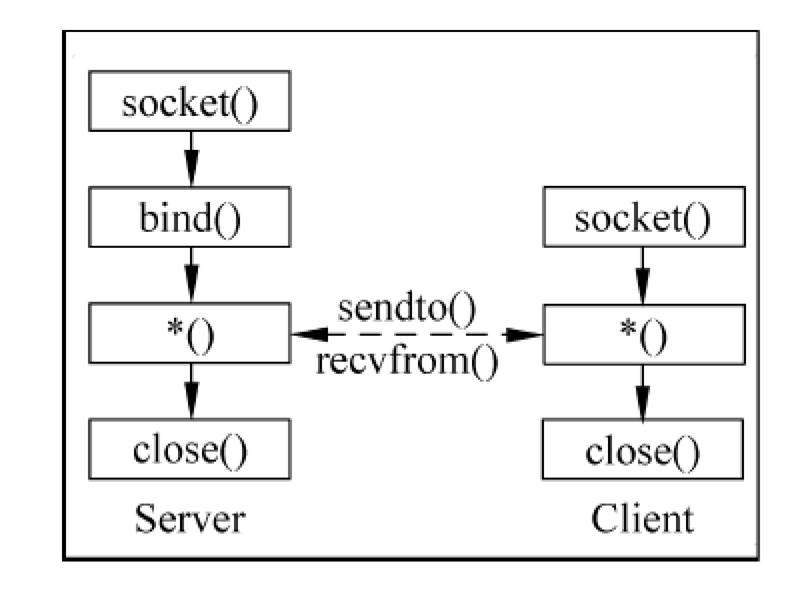


图 18-4　基于 socket 的 UDP 程序架构

基于套接字的 UDP Server 的网络编程一般包括以下基本步骤。

(1) 创建 socket 对象。

(2) 将 socket 绑定到指定地址上。

(3) 服务器和客户机通过 send()和 recv()方法通信（传输数据）。

(4) 传输结束，调用 socket 的 close()方法关闭连接。

其中，第(3)步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

基于套接字的 UDP Client 的网络编程一般包括以下基本步骤。

（1）创建 socket 对象。

（2）客户机和服务器通过 send()和 recv()方法通信(传输数据)。

（3）传输结束，调用 socket 的 close()方法关闭连接。

其中，第(2)步是实现程序功能的关键步骤，其他步骤在各种程序中基本相同。

18.2.2 创建 socket 对象

用户可以使用 socket 对象的构造函数创建一个 socket 对象，其语法形式如下：

```
socket(family = 2, type = 1, proto = 0, fileno = None)
```

各参数的意义如下。

- family：地址序列，默认为 AF_INET(2，socket 模块中的常量)，对应于 IPv4；而 AF_UNIX 对应于 UNIX 的进程间通信，AF_INET6 对应于 IPv6。
- type：socket 类型，默认为 SOCK_STREAM 对应于 TCP 流套接字；而 SOCK_DGRAM 对应于 UDP 数据报套接字，SOCK_RAW 对应于 raw 套接字。

例如：

```
>>> import socket
>>> s1 = socket.socket()                                          #创建用于 TCP 通信的套接字
>>> s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)         #创建用于 TCP 通信的套接字
>>> s3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)          #创建用于 UDP 通信的套接字
```

18.2.3 将服务器端 socket 绑定到指定地址

1. 主机名和 IP 地址

socket 模块包含下列若干函数，用于获取主机名和 IP 地址等信息。

```
• socket.gethostname()                        #返回主机名
• socket.gethostbyname(hostname)              #返回主机名的 IP 地址
• socket.gethostbyname_ex(hostname)           #返回扩展信息元组:(hostname, aliaslist, ipaddrlist)
• getfqdn([name])                             #返回全限定名称
• gethostbyaddr(ip_address)                   #返回 IP 地址的主机信息元组:(hostname, aliaslist,
                                              #ipaddrlist)
• getservbyname(servicename[, protocolname])  #返回服务所使用的端口号
```

例如：

```
>>> socket.gethostname()                      #返回本机的主机名,输出:'PC201607031137'
>>> socket.gethostbyname('www.baidu.com')     #返回百度的 IP 地址,输出:'119.75.216.20'
>>> socket.gethostbyname_ex('www.baidu.com')
('www.a.shifen.com', ['www.baidu.com'], ['119.75.216.20', '119.75.213.61'])
>>> socket.getservbyname('http','tcp')        #返回 http 服务所使用的端口号,输出:80
```

2. 绑定 socket 对象到 IP 地址

在创建服务器端 socket 对象后，必须把对象绑定到某个 IP 地址，然后客户机才可以与之连接。用户可以使用对象方法 bind()将 socket 绑定到指定的 IP 地址上，其语法形式如下：

```
sock.bind(address)
```

其中，address 是要绑定的 IP 地址，对应 IPv4 的地址为一个元组：

```
(主机名或 IP 地址, 端口号)
```

例如：

```
>>> sock = socket.socket()
```

```
>>> sock.bind(('localhost', 8000))              #绑定到本机 localhost 端口号 8000
>>> sock1 = socket.socket()
>>> sock1.bind((socket.gethostname(), 8001)) #绑定到本机端口号 8001
>>> sock2 = socket.socket()
>>> sock2.bind(('127.0.0.1', 8002))             #绑定到本机 127.0.0.1 端口号 8002
```

18.2.4　服务器端 socket 开始侦听

在创建服务器端 socket 对象并绑定到 IP 地址后，可以使用对象方法 listen()和 accept()进行侦听和接收连接，其语法形式如下：

```
sock.listen(backlog)
```

其中，backlog 是最多连接数，至少为 1，在接到连接请求后，这些请求必须排队，如果队列已满，则拒绝请求。例如：

```
>>> sock = socket.socket()
>>> sock.bind(('localhost', 8000))            #绑定到本机 localhost 端口号 8000
>>> sock.listen(5)                            #开始侦听，连接队列长度为 5
```

18.2.5　连接和接收连接

客户机端 socket 对象通过 connect()方法尝试建立到服务器端 socket 对象的连接，其语法形式如下：

```
client_sock.connect(address)          #client_sock 连接到绑定到 address 的服务器端 socket 对象
```

其中，address 是要连接的服务器端 socket 对象绑定的 IP 地址，对应 IPv4 的地址为一个元组。

服务器端 socket 对象通过 accept()方法进入'waiting'(阻塞)状态。当接收到来自客户的请求连接时，accept()方法建立连接并返回服务器。accept()方法返回一个含有两个元素的元组，即(clientsocket, address)，其中，clientsocket 是新建的 socket 对象，服务器通过它与客户通信；address 为对应的 IP 地址：

```
clientsocket, address = server_sock.accept()
```

18.2.6　发送和接收数据

对于面向连接的 TCP 通信程序，在客户机和服务器建立连接后，通过 socket 对象的 send()和 recv()方法分别发送和接收数据，语法形式如下：

- **send(bytes)**　　　　#发送数据 bytes，返回实际发送的字节数
- **sendall(bytes)**　　　#发送数据 bytes，持续发送；成功返回 None，否则出错
- **recv(bufsize)**　　　　#接收数据，返回接收到的数据：bytes 对象

其中，bytes 为字节序列；bufsize 为一次接收的数据的最大字节数。

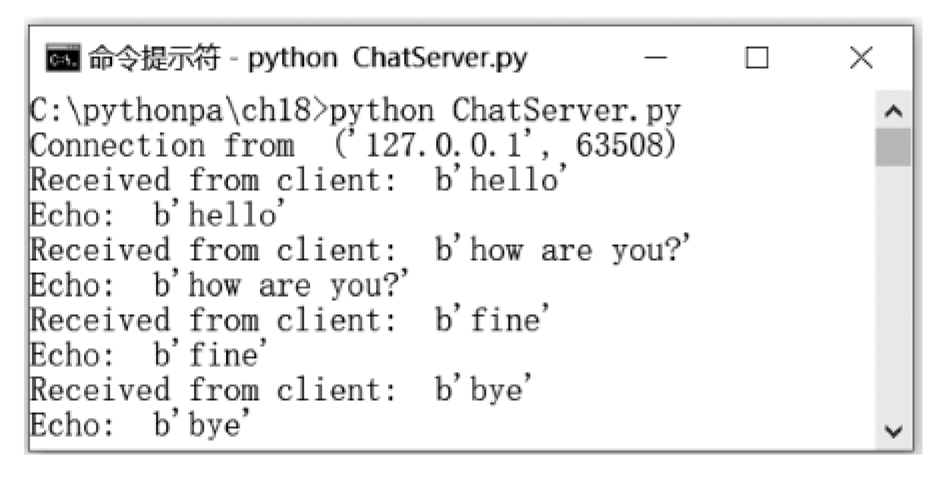
对于非面向连接的 UDP 通信程序，客户机和服务器不需要预先建立连接，直接通过 socket 对象的 sendto()指定发送目标地址参数，recvfrom()方法返回接收的数据以及发送源地址，语法形式如下：

- **sendto(bytes, address)**　　　#发送数据 bytes 到地址 address，返回实际发送的字节数
- **recvfrom(bufsize[, flags])**　　#接收数据，返回元组：(bytes, address)

其中，bytes 为字节序列；address 是发送的目标地址；bufsize 为一次接收的数据的最大字节数。

18.2.7 简单 TCP 程序：Echo Server

基于 TCP 的 Echo Server 包括服务器、客户机两个部分，即服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket，并绑定到某个“IP 地址：端口号”上，然后侦听 listen，并使用阻塞方法 accept()以等待客户机连接请求；客户机创建一个 socket，并建立到服务器的连接；客户机循环接收用户数据并发送数据到服务器，服务器接收数据后回送（Echo）给客户机。当客户机输入空数据时，关闭 socket 并终止运行；当服务器接收到空数据时，关闭 socket 并终止运行。其运行效果如图 18-5 所示。



(a) 服务端应用程序

```
命令提示符 - python ChatClient.py
C:\pythonpa\ch18>python ChatClient.py
>hello
Received from server: b'hello'
>how are you?
Received from server: b'how are you?'
>fine
Received from server: b'fine'
>bye
Received from server: b'bye'
>
```

(b) 客户机应用程序

图 18-5 简单 TCP 程序：Echo Server

注意：读者可以在单机上同时运行服务端应用程序和客户机应用程序，但建议在不同的计算机上运行服务端应用程序和客户机应用程序。如果服务端应用程序在其他计算机上运行，请把代码中创建 socket 对象时的服务器地址“127.0.0.1”修改为对应服务器的计算机地址。

【例 18.1】 简单 TCP 程序（ChatServer.py）：Echo Server。服务端应用程序 ChatServer。

```
import socket                                               #导入 socket 模块
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                            #创建服务器 socket
serversocket.bind(('127.0.0.1', 8000))                      #绑定到 IP 地址和端口号
serversocket.listen(1)                                      #开始侦听，队列长度为 1
clientsocket, clientaddress = serversocket.accept() #使用阻塞方法 accept()以等待客户机连接请求
print('Connection from ', clientaddress)                    #接收客户机请求后输出客户机的信息
while 1:                                                    #循环以接收和回送客户机数据
    data = clientsocket.recv(1024)                          #接收数据
    if not data: break                                      #接收到空数据时终止循环
    print('Received from client: ', repr(data))   #输出接收到的数据，用 repr()函数转换为字符串
    print('Echo: ', repr(data))                             #输出发送到客户机的数据的信息
    clientsocket.send(data)                                 #回送数据到客户机
clientsocket.close()                                        #关闭客户机 socket
serversocket.close()                                        #关闭服务器 socket
```

【例 18.2】 简单 TCP 程序（ChatClient.py）：Echo Server。客户机应用程序 ChatClient。

```
import socket                                               #导入 socket 模块
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                                                            #创建客户机 socket
clientsocket.connect(('127.0.0.1', 8000))                   #连接到服务器
while 1:                                                    #循环以接收用户输入，并发送到服务器，接收服
                                                            #务器的回送数据
    data = input('>')                                       #接收用户输入的数据
    clientsocket.send(data.encode())                        #把数据转换为 bytes 对象，并发送到服务器
    if not data: break                                      #如果数据为空，终止循环
```

```
    newdata = clientsocket.recv(1024)                 #接收服务器的回送数据
    print('Received from server: ', repr(newdata))    #输出接收到的数据
clientsocket.close()                                  #关闭客户机 socket
```

18.2.8　简单 UDP 程序：Echo Server

基于 UDP 的网络程序是无连接的，服务器和客户端不需要实现建立连接，发送数据时直接指定地址参数，接收数据时同时返回地址，通信双方地位平等，传输无法保证对方能够接收到数据报。

基于 UDP 的 Echo Server 包括服务器和客户机两个部分，即服务端应用程序和客户机应用程序。服务端应用程序创建一个 socket，并绑定到某个"IP 地址：端口号"上，然后循环使用 recvfrom()接收数据(返回数据和客户机地址)，并使用 sendto()回送数据到客户机地址；客户机创建一个 socket，然后循环使用 sendto()发送用户输入的数据到服务器，并接收服务器回送的数据。当客户机输入空数据时，关闭 socket 并终止运行；当服务器接收到空数据时，关闭 socket 并终止运行。其运行效果如图 18-6 所示。

```
命令提示符 - python ChatServerUDP.py
C:\pythonpa\ch18>python ChatServerUDP.py
Received from client:  ('127.0.0.1', 62743) b'hello'
Echo:  b'hello'
Received from client:  ('127.0.0.1', 62743) b'how are you?'
Echo:  b'how are you?'
Received from client:  ('127.0.0.1', 62743) b'fine'
Echo:  b'fine'
Received from client:  ('127.0.0.1', 62743) b'bye'
Echo:  b'bye'
```

(a) 服务端应用程序

```
命令提示符 - python ChatClientUDP.py
C:\pythonpa\ch18>python ChatClientUDP.py
>hello
Received from server:  (b'hello', ('127.0.0.1', 8000))
>how are you?
Received from server:  (b'how are you?', ('127.0.0.1', 8000))
>fine
Received from server:  (b'fine', ('127.0.0.1', 8000))
>bye
Received from server:  (b'bye', ('127.0.0.1', 8000))
>
```

(b) 客户机应用程序

图 18-6　简单 UDP 程序：Echo Server

注意：读者可以在单机上同时运行服务端应用程序和客户机应用程序。但建议在不同的机器上运行服务端应用程序和客户机应用程序。如果服务端应用程序在其他机器上运行，请把代码中创建 Socket 对象时的服务器地址"127.0.0.1"修改为对应服务器的机器地址。

【例 18.3】　简单 UDP 程序：Echo Server。服务端应用程序 ChatServerUDP。

```
#ChatServerUDP.py
import socket                                         #导入 socket 模块
serversocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                                                      #创建服务器 socket
serversocket.bind(('127.0.0.1', 8000))                #绑定到 IP 地址和端口号
while 1:                                              #循环以接收和回送客户机数据
    data, address = serversocket.recvfrom(1024)       #接收数据，返回数据和客户机地址
    if not data: break;                               #接收到空数据时终止循环
    print('Received from client: ', address, repr(data))
                                                      #输出接收到的数据，用 repr()函数转换为字符串
    print('Echo: ', repr(data))                       #输出发送到客户机的数据的信息
    serversocket.sendto(data, address)                #发送数据到客户机
serversocket.close()                                  #关闭服务器 socket
```

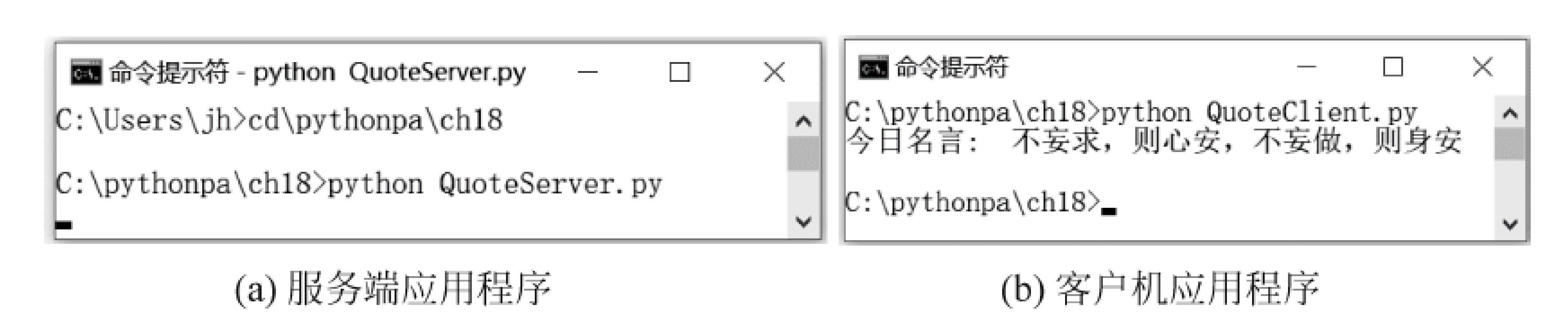
【例 18.4】　简单 UDP 程序：Echo Server。客户机应用程序 ChatClientUDP。

```
#ChatClientUDP.py
import socket                                         #导入 socket 模块
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                                                      #创建客户机 socket
```

```
while 1:                                          #循环以接收用户输入,并发送到服务器,接收服
                                                  #务器的回送数据
    data = input('>')                             #接收用户输入的数据
    clientsocket.sendto(data.encode(),('127.0.0.1', 8000))
                                                  #把数据转换为 bytes 对象,并发送到服务器
    if not data: break                            #如果数据为空,终止循环
    newdata = clientsocket.recvfrom(1024)         #接收服务器的回送数据
    print('Received from server: ', repr(newdata))  #输出接收到的数据
clientsocket.close()                              #关闭客户机 socket
```

18.2.9 UDP 程序：Quote Server

Quote Server 实现 Quote of the day(每日名言)功能：客户机发送一个数据报到 Quote 服务器(相当于请求)；服务器接收来自客户机的数据报(请求)；服务器从格言列表中读取一句名言,并作为数据报发送给客户机；客户机接收 Quote 服务器的数据报(包含一句名言),并显示该名言。其运行结果如图 18-7 所示。



(a) 服务端应用程序　　(b) 客户机应用程序

图 18-7 UDP 程序：Quote Server

注意：读者可以在单机上同时运行服务端应用程序和客户机应用程序,但建议在不同的计算机上运行服务端应用程序和客户机应用程序。

【例 18.5】 UDP 程序(QuoteServer.py)：实现 Quote of the day(每日名言)功能。服务端应用程序 QuoteServer。

```
import socket, random                             #导入 socket 和 random 模块
quotes = ['不妄求,则心安,不妄做,则身安','多门之室生风,多言之人生祸','人之心胸,多欲则窄,寡欲
         则宽','三人行,必有我师','滴水穿石,磨杵成针','是非天天有,不听自然无','积德为产业,强
         胜于美宅良田']
serversocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                                                  #创建服务器 socket
serversocket.bind(('127.0.0.1', 8002))            #绑定到 IP 地址和端口号
while 1:                                          #循环以接收和回送客户机数据
    data, address = serversocket.recvfrom(1024)   #接收数据,返回数据和客户机地址
    quote = random.choice(quotes)                 #从 Quotes 列表中随机选择一个项目
    serversocket.sendto(quote.encode(), address)  #把数据转换为 bytes 对象, 并发送数据到客户机
serversocket.close()                              #关闭服务器 socket
```

【例 18.6】 UDP 程序(QuoteClient.py)：实现 Quote of the day(每日名言)功能。客户机应用程序 QuoteClient。

```
import socket                                     #导入 socket 模块
clientsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                                                  #创建客户机 socket
clientsocket.sendto(b'hello',('127.0.0.1', 8002)) #把数据转换为 bytes 对象,并发送到服务器
newdata, address = clientsocket.recvfrom(1024)    #接收服务器的回送数据
print('今日名言: ', newdata.decode())              #接收到数据解码为字符串,并输出
clientsocket.close()                              #关闭客户机 socket
```

18.3　基于 urllib 的网络编程

urllib 模块包含 4 个子模块，即 urllib.request（打开和读取 URL）、urllib.parse（解析 URL）、urllib.error（urllib.request 引发的异常）和 urllib.robotparser（解析 robots.txt 文件）。

18.3.1　打开和读取 URL 网络资源

使用 urllib.request 模块中的 urlopen()函数可以打开 URL，其语法形式如下：

```
urllib.request.urlopen(url, data = None)            #打开指定的 URL
```

其中，url 可以为字符串或 Request 对象；可选参数 data 是向服务器传送的数据。对于 HTTP/HTTPS 协议，urlopen()返回 response 对象（file-like 的对象），可以从中读取和输出内容。

【例 18.7】　打开和读取 URL 网络资源示例。

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.baidu.com')
                                                  #打开 URL 资源
>>> print(f.read(200))                            #读取 200 个字节，返回 bytes 对象并输出
b'<!DOCTYPE html>\r\n<html>\r\n\t<head>\r\n\t\t<meta http-equiv="content-type"
content="text/html;charset=utf-8">\r\n\t\t<meta http-equiv="X-UA-Compatible" content
="IE=Edge">\r\n\t\t<meta content="never" name="referrer"'
>>> f = urllib.request.urlopen('http://www.baidu.com')
                                                  #重新打开 URL 资源
>>> print(f.read(200).decode())                   #读取 200 个字节，返回 bytes 对象，转换为
                                                  #字符串并输出
>>> with urllib.request.urlopen('http://www.baidu.com/') as f:
                                                  #重新打开 URL 资源
        print(f.read(200).decode('utf-8'))        #读取返回 bytes 对象转换为字符串并输出
```

18.3.2　创建 Request 对象

urllib.request 模块中 Request 对象的构造函数如下：

```
urllib.request.Request(url, data = None, headers = {}, origin_req_host = None, unverifiable =
False, method = None)
```

其中，url 为字符串；可选参数 data（需要编码为 utf-8）是向服务器传送的数据；headers 是字典，为传递的 header 数据。

Request 对象 request 包含下列主要属性和方法。

- request.full_url：Request 对象 request 的 URL。
- request.host：主机和端口号。
- request.data：向服务器传送的数据。
- request.add_data(data)：添加向服务器传送的数据。
- request.add_header(key, val)：添加向服务器传送的 header。

【例 18.8】　Request 对象示例（RequestTest.py）。

```
import urllib.request                             #导入 urllib.request 模块
def getURLInfo(url, data, headers):
    req = urllib.request.Request(url, data, headers) #创建 Request 对象
    print('Full url:', req.full_url)              #URL
```

```
    print('Host:', req.host)                       #主机和端口号
    print('Data:', req.data)                       #向服务器传送的数据
#测试代码
if __name__ == '__main__':
    url = 'http://www.baidu.com/s'
    values = {'wd':'python'}
    data = urllib.parse.urlencode(values)
    data = data.encode(encoding = 'UTF8')
    headers = {'User-Agent':'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}
    getURLInfo(url, data, headers)
```

程序运行结果如下。

```
Full url: http://www.baidu.com/
Host: www.baidu.com
Data: b'wd = python
```

18.4 基于 http 的网络编程

http 模块包含 4 个子模块，即 http.client(低级别的 HTTP 协议客户端，高级别的 URL 打开则使用 urllib.request)、http.server(基于 socketserver 的 HTTP 服务器类)、http.cookies(使用 cookies 实现状态管理的工具)和 http.cookiejar(提供 cookies 的持久性)。

一般不直接使用 http.client 模块访问 HTTP 服务器，建议使用 urllib.request 模块。事实上，urllib.request 模块使用 http.client 模块处理包含 http 和 https 的 URL。

18.5 基于 ftplib 的网络编程

ftplib 模块包含 FTP 对象，实现了 FTP 客户端协议，用于访问 FTP 服务器。使用 ftplib 模块可以编写批量处理 FTP 服务器内容的程序。

18.5.1 创建 FTP 对象

ftplib 模块中 FTP 对象的构造函数如下：

ftplib.FTP(host = '', user = '', passwd = '', acct = '', timeout = None, source_address = None)

其中，host 为 FTP 服务器；user、passwd 和 acct 为用于登录的用户名、密码和账户(账户信息大部分 FTP 服务器不支持)；timeout 为超时时间；source_address 为元组(host，port)。在创建 FTP 对象时，如果指定了 host，则自动调用对象方法 connect(host)连接到 FTP 服务器；如果指定了 user，则自动调用对象方法 login(user，passwd，acct)登录到 FTP 服务器。

FTP 对象 ftp 包含下列主要属性和方法(通常对应于 FTP 命令)。

- ftp.set_debuglevel(level)：设置调试级别为 0(默认值，无调试信息)、1(基本调试信息)或 2 以上(详细调试信息)。
- ftp.connect(host='', port=0, timeout=None, source_address=None)：连接到 FTP 服务器。
- ftp.getwelcome()：返回欢迎信息。
- ftp.login(user='anonymous', passwd='', acct='')：登录到 FTP 服务器。
- ftp.abort()：终止传输。

- ftp. retrbinary(cmd, callback, blocksize=8192, rest=None): 下载文件(二进制传输模式)。
- ftp. retrlines(cmd, callback=None): 下载文件(文本传输模式)。
- ftp. storbinary(cmd, file, blocksize=8192, callback=None, rest=None): 上传文件(二进制传输模式)。
- ftp. storlines(cmd, file, callback=None): 上传文件(文本传输模式)。
- ftp. set_pasv(boolean): 设置传输模式,其中 True 为被动模式,False 为主动模式。
- ftp. cwd(pathname): 切换当前目录。
- ftp. mkd(pathname): 创建目录。
- ftp. pwd(): 打印当前目录。
- ftp. rmd(dirname): 删除目录。
- ftp. mlsd(path="", facts=[]): 列目录,取代旧版本中的 dir()方法。
- ftp. rename(fromname, toname): 文件重命名。
- ftp. delete(filename): 删除文件。
- ftp. size(filename): 获取文件大小。
- ftp. quit(): 退出(polite way)。
- ftp. close(): 关闭。若退出或关闭 FTP 对象,则不能继续操作 FTP 对象。

【例 18.9】 创建 FTP 对象示例。

```
>>> from ftplib import FTP
>>> ftp = FTP("ftp1.at.proftpd.org")
>>> ftp.login()                  # '230 Anonymous access granted, restrictions apply'
>>> ftp.dir()
 -rw-rw-r--      1 ftp     ftp     451 Jul      1     2005 README.MIRRORS
drwxrwxr-x       3 ftp     ftp     4096 Jul     1     2005 devel
drwxrwxr-x       3 ftp     ftp     4096 Dec     2     2010 distrib
drwxrwxr-x       4 ftp     ftp     4096 Jul     1     2005 historic
>>> ftp.cwd('devel')             # '250 CWD command successful'
>>> ftp.dir()
drwxrwxr-x       2 ftp     ftp     4096 Jul 10 00:07 source
```

18.5.2 创建 FTP_TLS 对象

ftplib 模块中的 FTP_TLS 对象继承于 FTP 对象,FTP_TLS 对象的构造函数为:

ftplib.FTP_TLS(host = '', user = '', passwd = '', acct = '', keyfile = None, certfile = None, context = None, timeout = None, source_address = None)

其中,keyfile 和 certfile 为证书文件; context 为 ssl. SSLContext。其他参数的意义同 FTP 构造函数。

FTP 对象 ftps 增加了下列主要属性和方法。

- ftps. ssl_version: SSL 版本(默认为 TLSv1)。
- ftps. auth(): 设置加密控制连接。
- ftps. ccc(): 取消控制通道,回到明文传输。
- ftps. prot_p(): 设置为加密传输。
- ftps. prot_c(): 设置为明文传输。

【例 18.10】 创建 FTP_TLS 对象示例。

```
>>> from ftplib import FTP_TLS
>>> ftps = FTP_TLS('ftp.python.org')
>>> ftps.login()                       #匿名登录安全控制通道
```

```
>>> ftps.prot_p()                       #安全数据连接(加密传输)
>>> ftps.retrlines('LIST')              #罗列目录清单
total 9
drwxr-xr-x    8 root    wheel    1024 Jan 3 1994 .
drwxr-xr-x    8 root    wheel    1024 Jan 3 1994 ..
drwxr-xr-x    2 root    wheel    1024 Jan 3 1994 bin
drwxr-xr-x    2 root    wheel    1024 Jan 3 1994 etc
d-wxrwxr-x    2 ftp     wheel    1024 Sep 5 13:43 incoming
drwxr-xr-x    2 root    wheel    1024 Nov 17 1993 lib
drwxr-xr-x    6 1094    wheel    1024 Sep 13 19:07 pub
drwxr-xr-x    3 root    wheel    1024 Jan 3 1994 usr
-rw-r--r--    1 root    root     312 Aug 1 1994 welcome.msg
'226 Transfer complete.'
>>> ftps.quit()                        #退出
```

18.6　基于 poplib 和 smtplib 的网络编程

poplib 模块提供了对 POP3 协议的支持，smtplib 模块提供了对 SMTP 协议的支持，使用 poplib 和 smtplib 可以实现接收和发送邮件的功能。

18.6.1　使用 poplib 接收邮件

poplib 模块中的 POP3 对象用于连接到 POP3 服务器，其构造函数为：

poplib.POP3(host, port = POP3_PORT[, timeout])

其中，host 和 port 为 POP3 服务器及其端口号；timeout 为超时时间。

POP3 对象 pop3 包含下列主要属性和方法。

- pop3.set_debuglevel(level)：设置调试级别为 0（默认值，无调试信息）、1（基本调试信息）或 2 以上（详细调试信息）。
- pop3.user(username)：发送 user 命令，响应需要密码。
- pop3.user(username)：发送密码，返回邮件数和邮箱大小，锁定邮箱直至调用 quit()。
- pop3.getwelcome()：返回欢迎信息。
- pop3.stat()：返回邮箱状态，结果为元组（message count，mailbox size）。
- pop3.list([which])：返回邮件列表。
- pop3.retr(which)：接收邮件，并设置其状态为已读。
- pop3.dele(which)：设置邮件删除标记，调用 quit()时删除邮件。
- pop3.rset()：清除邮件删除标记。
- pop3.noop()：空操作，用于保持连接状态。
- pop3.quit()：注销退出，释放邮箱锁定，释放连接。
- pop3.top(which，howmuch)：接收邮件部分内容。
- pop3.uidl(which=None)：返回邮件摘要列表。

【例 18.11】　pop3 示例（pop3.py）。

```
import getpass, poplib
host = 'YourPop3Host'                   #POP3 服务器的主机名或 IP 地址,运行时需修改为对应的值
port = 110                              #POP3 服务器的端口号,默认为 110,运行时需修改为对应的值
pop3 = poplib.POP3(host, port=port)     #创建 POP3 对象
pop3.user(getpass.getuser())            #用户名
pop3.pass_(getpass.getpass())           #密码
numMessages = len(pop3.list()[1])       #邮件数
```

```
for i in range(numMessages):          #接收邮件
    for j in pop3.retr(i+1)[1]:
        print(j)
```

18.6.2　使用 smtplib 发送邮件

poplib 模块中的 SMTP 对象用于连接到 SMTP/ESMTP 服务器，其构造函数为：

smtplib.SMTP(host='', port=0, local_hostname=None[, timeout], source_address=None)

其中，host 和 port 为 SMTP/ESMTP 服务器及其端口号；local_hostname 为本地主机；timeout 为超时时间。如果指定了 host 和 port，则自动调用对象方法 connect()连接到 SMTP/ESMTP 服务器。

SMTP 对象 smtp 包含下列主要属性和方法。

- smtp.set_debuglevel(level)：设置调试级别为 0(默认值，无调试信息)、1(基本调试信息)或 2 以上(详细调试信息)。
- smtp.docmd(cmd, args='')：发送命令到服务器。
- smtp.connect(host='localhost', port=0)：连接到服务器。
- smtp.login(user, password)：登录到服务器。
- smtp.sendmail(from_addr, to_addrs, msg, mail_options=[], rcpt_options=[])：发送邮件。
- smtp.quit()：注销退出，释放邮箱锁定，释放连接。

【例 18.12】　smtp 示例(smtp.py)。

```
import smtplib
def prompt(prompt):
    return input(prompt).strip()
fromaddr = prompt("From: ")
toaddrs = prompt("To: ").split()
print("输入信息,^D (Unix) or ^Z (Windows)结束输入:")
#添加 From: 和 To: 头信息
msg = ("From: %s\r\nTo: %s\r\n\r\n" % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line
print("信息长度为:", len(msg))
server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

18.7　复　习　题

一、填空题

1. TCP/IP 协议模型把 TCP/IP 协议族分成 4 个层次，即______、______、______和______。

2. Internet 采用一种全局通用的地址格式为网络中的每一台主机都分配一个唯一的地址，称为________。

3. Internet 使用________来管理计算机域名与 IP 地址的对应关系。

4. IP 地址用来标识 Internet 上的主机，而位于 Internet 主机上的资源（例如各种文档、图像等）则通过________来标识。

5. TCP/IP 协议的传输层包含两个传输协议，即面向连接的________和非面向连接的________。

6. 在创建服务器端 socket 对象并绑定到 IP 地址后，可以使用________和________对象方法进行侦听和接收连接。

7. 客户机端 socket 对象通过________方法尝试建立到服务器端 socket 对象的连接。

二、思考题

1. 如何用 Python 接收和发送邮件？
2. 基于套接字的 TCP Server 的网络编程一般包括哪些基本步骤？
3. 基于套接字的 TCP Client 的网络编程一般包括哪些基本步骤？
4. 基于套接字的 UDP Server 的网络编程一般包括哪些基本步骤？
5. 基于套接字的 UDP Client 的网络编程一般包括哪些基本步骤？
6. 对于面向连接的 TCP 通信程序，客户机和服务器建立连接后如何发送和接收数据？
7. 对于非面向连接的 UDP 通信程序，客户机和服务器间如何发送和接收数据？
8. urllib 模块包含哪几个子模块，分别实现什么功能？
9. http 模块包含哪几个子模块，分别实现什么功能？
10. 如何实现基于 ftplib 的网络编程？
11. 如何使用 poplib 模块和 smtplib 模块实现邮件接收和发送功能？

18.8 上机实践

完成本章中的例 18.1～例 18.12，熟悉 Python 网络编程和通信。

18.9 案例研究：网络爬虫案例

网络爬虫是通过跟踪超链接系统访问 Web 页面的程序，每次访问一个网页时会分析网页内容，提取结构化数据信息。

本章案例研究通过 4 种常用的网络爬虫的实现案例帮助读者深入了解 Python 网络应用程序的开发流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 19 章

并行计算：进程、线程和协程



视频讲解

用户可以使用 urllib、http、ftplib、poplib、smtplib 等模块针对特定网络协议编程，还可以使用扩展库进行网络编程。

线程能够执行并发处理，即同时执行多个操作。例如，使用线程处理同时监视用户输入，并执行后台任务，以及处理并发输入流。

19.1 并行处理概述

19.1.1 进程、线程和协程

进程是操作系统中正在执行的不同应用程序的一个实例，操作系统把不同的进程分离开来。每一个进程都有自己的地址空间，一般包括文本区域、数据区域和堆栈区域。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

在现代多核 CPU 计算机中，使用进程可以并行处理多个任务，从而提高计算密集任务程序的性能。

线程是进程中的一个实体，是被操作系统独立调度和分派处理器时间的基本单位。线程一般由线程 ID、当前指令指针、寄存器集合和堆栈组成，即一组用于调度的该线程上下文的结构；线程与同属一个进程的其他线程共享进程所拥有的全部资源。线程又称为轻量级进程(Light Weight Process，LWP)。支持抢先多任务处理的操作系统可以实现多个进程中的多个线程同时执行的效果：在需要处理器时间的线程之间分割可用处理器时间，并轮流为每个线程分配处理器时间片(时间片的长度取决于操作系统和处理器数目)，由于每个时间片都很小，所以多个线程看起来似乎在同时执行。

线程使程序能够执行并发处理，因而特别适合需要同时执行多个操作的场合。例如，使用一个线程来执行复杂的后台计算任务，使用另一个线程来监视用户输入，以提高系统的用户响应性能；使用高优先级线程管理时间关键的任务，使用低优先级线程执行其他任务，以区分具有不同优先级的任务；为服务器应用程序创建包含多个线程的线程池，以及处理并发的客户端请求。线程适用于 IO 密集任务的场合，能有效避免程序 IO 阻塞，可以提高程序的响应性能。

协程(Coroutine)又称微线程、纤程，协程不是进程或线程，其执行过程更类似于函数调用。在 Python 的 asyncio 模块实现的异步 IO 编程框架中，协程是对使用 async 关键字定义的异步函数的调用。一个进程包含多个线程，同样，一个程序可以包含多个协程。多个线程相对独立，线程有自己的上下文，切换受系统控制；同样，多个协程也相对独立，协程也有自己的上

下文,但是其切换由程序自己控制。

协程适合于异步 IO 编程的场合,能有效提高 IO 的吞吐效率。

虽然多进程、多线程和协程处理可以解决用户响应性能和多任务的问题,但同时引入了资源共享和同步等问题。例如,过多的线程将占用大量的资源和处理器调度时间,从而影响运行性能;为了避免对共享资源的访问冲突,必须对共享资源进行同步或控制处理,因而有可能导致死锁;使用多线程控制代码执行非常复杂,并可能产生许多错误。

19.1.2 Python 语言与并行处理相关模块

Python 标准库中包括下列与并行处理相关的模块。

- _thread 和_dummy_thread 模块:底层低级线程 API。
- threading 模块:线程及其同步处理。
- multiprocessing 模块:多进程处理和进程池。
- concurrent.futures 模块:启动并行任务。
- queue 模块:线程安全队列,用于线程或进程间的信息传递。
- asyncio 模块:异步 IO、事件循环、协程和任务处理。

19.2 基于线程的并发处理

19.2.1 threading 模块概述

Python 标准库模块 threading 提供了与线程相关的操作,例如创建线程、启动线程、线程同步等。

通过创建 threading.Thread 对象实例可以创建线程;通过调用 Thread 对象的 start()方法可以启动线程,也可以创建 Thread 的派生类,重写 run()方法,然后通过创建其对象实例来创建线程;通过线程对象的 daemon 属性可以设置线程为用户线程或 daemon 线程。

当多个线程调用单个对象的属性和方法时,一个线程可能会中断另一个线程正在执行的任务,使该对象处于一种无效状态,因此必须针对这些调用进行同步处理。

Python 语言提供了多种线程同步处理解决方案,例如 Lock/RLock 对象、Condition 对象、Semaphore 对象、Event 对象、Barrier 对象等。

使用 Python 标准模块 queue 提供的线程安全队列也可以方便地实现生产者和消费者线程之间的同步。

值得注意的是,由于历史原因,Python 语言经典实现 CPython 使用了 GIL(Global Interpreter Lock,全局解释器锁)。每个线程在执行的过程中都需要先获取 GIL,以保证同一时刻只有一个线程可以执行代码。当 IO 操作等可能会引起阻塞的调用之前,或者执行时间达到一定阈值后,当前线程会释放 GIL,其他线程获得 GIL 并执行。

也就是说,在 CPython 中线程并没有实现真正意义上的并行处理,即使在多核 CPU 的情况下,也只能同时执行一个线程,其通过时间片的方法使得多个线程看起来并行运行,但不能真正发挥多核 CPU 的运算能力。在处理像 IO 操作等可能引起阻塞的这类任务(例如网页爬取)的时候,多线程会比单线程快;然而在处理像科学计算等需要持续使用 CPU 的计算密集任务的时候,单线程反而会比多线程快。因此,建议在 IO 密集型(读取文件、读取网络套接字

等)任务中使用多线程，在计算密集型(大量消耗 CPU 的数学与逻辑运算)任务中使用多进程。

如果要真正使用多核 CPU 进行并行计算，可以使用 Python 标准库中的 multiprocessing 模块，也可以使用其他 Python 实现。

19.2.2 使用 Thread 对象创建线程

threading 模块封装了_thread 模块，并提供更多功能。虽然可以使用_thread 模块中的 start_new_thread()函数创建线程，但一般建议使用 threading 模块。

通过创建 Thread 的对象可以创建线程：

```
Thread(target = None, name = None, args = (), kwargs = {})          #构造函数
```

其中，target 是线程运行的函数；name 是线程的名称；args 和 kwargs 是传递给 target 的参数元组和命名参数字典。

通过调用 Thread 对象的 start()方法可以启动线程。Thread 对象的常用方法如下。

- t.start()：启动线程。
- t.is_alive()：判断线程是否活动。
- t.name：属性，线程名，对应于老版本的方法 getname()和 setname()。
- t.id：返回线程标识符。

threading 模块包含以下若干实用函数。

- threading.get_ident()：返回当前线程的标识符。
- threading.current_thread()：返回当前线程。
- threading.active_count()：返回活动的线程数目。
- threading.enumerate()：返回活动线程的列表。

【例 19.1】 直接使用 Thread 对象创建和启动新线程(td_thread.py)。

```
import threading, time, random
def timer(interval):
    for i in range(3):
        time.sleep(random.choice(range(interval)))          #随机睡眠 interval 秒
        thread_id = threading.get_ident()                   #获取当前线程标识符
        print('Thread:{0} Time:{1}'.format(thread_id, time.ctime()))
if __name__ == '__main__':
        t1 = threading.Thread(target = timer, args = (5,))  #创建线程
        t2 = threading.Thread(target = timer, args = (5,))  #创建线程
        t1.start(); t2.start()                              #启动线程
```

程序运行结果如下。

```
Thread:8292 Time:Wed Jul 11 09:56:44 2018
Thread:8292 Time:Wed Jul 11 09:56:44 2018
Thread:8292 Time:Wed Jul 11 09:56:44 2018
...
```

19.2.3 自定义派生于 Thread 的对象

通过声明 Thread 的派生类并重写对象的 run()方法，然后创建其对象实例，可以创建线程；通过对象的 start()方法可以启动线程，并自动执行对象的 run()方法。

【例 19.2】 通过声明 Thread 派生类创建和启动新线程(td_MyThread.py)。

```
import threading, time, random
class MyThread(threading.Thread):                              #继承 threading.Thread
    def __init__(self, interval):                              #构造函数
        threading.Thread.__init__(self)                        #调用父类构造函数
        self.interval = interval                               #对象属性
    def run(self):                                             #定义 run()方法
        for i in range(5):
            time.sleep(random.choice(range(self.interval)))   #随机睡眠 interval 秒
            thread_id = threading.get_ident()                  #获取当前线程标识符
            print('Thread:{0} Time:{1}\n'.format(thread_id, time.ctime()))
if __name__ == '__main__':
    t1 = MyThread(5)                                           #创建对象
    t2 = MyThread(5)                                           #创建对象
    t1.start(); t2.start()                                     #启动线程
```

程序运行结果如下。

```
Thread:10132 Time:Wed Jul 11 09:57:40 2018
Thread:10132 Time:Wed Jul 11 09:57:41 2018
Thread:8752 Time:Wed Jul 11 09:57:42 2018
Thread:8752 Time:Wed Jul 11 09:57:42 2018
…
```

19.2.4 线程加入

所谓线程加入(t.join()),就是让包含代码的线程(tc,即当前线程)"加入"到另外一个线程(t)的尾部。在线程(t)执行完毕之前,线程(tc)不能执行。线程加入的构造函数如下:

```
join(timeout = None)
```

其中,timeout 是超时参数,单位为秒。如果指定了超时,则线程 t 执行完毕或超时都可能使当前线程继续,此时可通过 t.is_alive()来判断线程是否终止。

注意:线程不能加入自己,否则将导致 RuntimeError,因为这样将导致死锁。线程也不能加入未启动的线程,否则将导致 RuntimeError。

【例 19.3】 线程 join 示例(td_join.py)。

```
import threading, time, random
class MyThread(threading.Thread):                              #继承 threading.Thread
    def __init__(self):                                        #构造函数
        threading.Thread.__init__(self)                        #调用父类构造函数
    def run(self):                                             #定义 run()方法
        for i in range(5):
            time.sleep(1)                                      #睡眠 1 秒
            t = threading.current_thread()                     #获取当前线程
            print('{0} at {1}\n'.format(t.name, time.ctime())) #打印线程名、当前时间
        print('线程 t1 结束')
def test():
    t1 = MyThread()                                            #创建线程对象
    t1.name = 't1'                                             #设置线程名称
    t1.start()                                                 #启动线程
    print('主线程开始等待线程(t1)2s'); t1.join(2)
    print('主线程等待线程(t1)2s 结束')
    print('主线程开始等待线程结束'); t1.join()
    print('主线程结束')
if __name__ == '__main__':
    test()
```

程序运行结果如下。

```
主线程开始等待线程(t1)2s
t1 at Wed Jul 11 09:59:18 2018
主线程等待线程(t1)2s 结束 t1 at Wed Jul 11 09:59:19 2018
主线程开始等待线程结束
t1 at Wed Jul 11 09:59:20 2018
t1 at Wed Jul 11 09:59:21 2018
t1 at Wed Jul 11 09:59:22 2018
线程 t1 结束
主线程结束
```

19.2.5　用户线程和 daemon 线程

线程可以分为用户线程和 daemon 线程。

用户线程(非 daemon 线程)是通常意义上的线程，应用程序运行即为主线程，在主线程中可以创建和启动新线程，默认为用户线程。只有当所有非 daemon 的用户线程(包括主线程)结束后应用程序才终止。

如果在主线程中创建新线程时设置其对象属性 daemon 为 True，则该线程为 daemon 线程。

```
t.daemon                    #属性,对应于老版本的方法 isDaemon()和 setDaemon()
```

其默认为 False，即非 daemon 线程。如果设置为 True，则为 daemon 线程。该属性必须在线程启动之前调用，否则将导致 RuntimeError，即已启动的线程不能改变其 daemon 属性。

daemon 线程又称守护线程，其优先级是最低的，一般为其他的线程提供服务。通常，daemon 线程体是一个无限循环。如果所有的非 daemon 线程都结束了，则 daemon 线程会自动终止。

【例 19.4】　用户线程和 daemon 线程示例(td_daemon.py)。程序运行结果如图 19-1 所示。



图 19-1　用户线程和 daemon 线程运行结果

```
import threading, time
class MyThread(threading.Thread):                    #继承 threading.Thread
    def __init__(self, interval):                    #构造函数
        threading.Thread.__init__(self)              #调用父类构造函数
        self.interval = interval                     #对象属性
    def run(self):                                   #定义 run()方法
        t = threading.current_thread()               #获取当前线程
        print('线程' + t.name + '开始')
        time.sleep(self.interval)                    #延迟 self.interval 秒
        print('线程' + t.name + '结束')
class MyThreadDaemon(threading.Thread):              #继承 threading.Thread
    def __init__(self, interval):                    #构造函数
        threading.Thread.__init__(self)              #调用父类构造函数
        self.interval = interval                     #对象属性
```

```
    def run(self):                                      #定义 run()方法
        t = threading.current_thread()                  #获取当前线程
        print('线程' + t.name + '开始')
        while True:
            time.sleep(self.interval)                   #延迟 self.interval 秒
            print('daemon 线程' + t.name + '正在运行')
        print('线程' + t.name + '结束')
def test():
    print('主线程开始')
    t1 = MyThread(5)                                    #创建线程对象
    t2 = MyThreadDaemon(1)                              #创建线程对象
    t1.name = 't1'; t2.name = 't2'                      #设置线程名称
    t2.daemon = True                                    #设置为 daemon
    t1.start()                                          #启动线程
    t2.start()
    print('主线程结束')
if __name__ == '__main__':
    test()
```

19.2.6 Timer 线程

在实际应用中,经常需要使用定时器去触发一些事件。

使用 Python 标准库 threading 中的 Timer 线程(Thread 的子类)可以很方便地实现定时器功能。Timer 对象包含的主要方法如下。

(1) Timer(interval, function, args=None, kwargs=None):构造函数,在指定时间 interval 后执行函数。

(2) start():启动线程,即启动计时器。

(3) cancel():取消计时器。

【例 19.5】 Timer 线程示例(td_timer.py)。

```
import threading
def f():
    print('Hello Timer!')                               #创建定时器,1 秒后运行
    global timer
    timer = threading.Timer(1, f)
    timer.start()
timer = threading.Timer(1, f)                           #创建定时器,1 秒后运行
timer.start()                                           #启动定时器
```

程序运行结果如下。

```
Hello Timer!
Hello Timer!
…
```

19.2.7 基于原语锁的简单同步

原语锁是同步原语,使用 threading 模块的 Lock 对象锁可以实现线程的简单同步。threading 模块的 RLock 是可重入的同步锁。

Lock 对象锁有两个状态,即 locked 和 unlocked(初始状态)。

线程可以使用 Lock 对象锁的 acquire()方法获得锁,此时锁进入 locked 状态。注意,每次只有一个线程可以获得锁。如果另一个线程试图使用 acquire()方法获得 locked 状态的锁,则锁就会被系统变为 blocked 状态,直到那个拥有锁的线程调用锁的 release()方法来释放锁,这

样锁就会进入 unlocked 状态。blocked 状态的线程会收到一个通知，并有权利获得锁。如果多个线程处于 blocked 状态，所有线程都会先解除 blocked 状态，然后系统选择一个线程来获得锁（具体是哪个线程获得锁与实现有关），其他的线程继续沉默（blocked）。

一般将需要实现线程同步的关键代码放置在 acquire()和 release()方法之间，即：

```
import threading
lock = threading.Lock()
lock.acquire()
…
lock.release()
```

Lock 对象支持 with 语句：

```
with some_lock:
    # do something...
```

等价于：

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Lock 不允许在同一线程中被多次 acquire，否则会产生死锁，而 RLock（可重入锁）没有这个限制。使用 RLock，acquire()和 release()必须成对出现，这样才能真正释放所占用的锁。例如，下面的代码片段会产生死锁：

```
import threading
lock = threading.Lock()
lock.acquire()
lock.acquire()                                      #会产生死锁
lock.release()
lock.release()
```

而下面的代码片段，则可以正常运行：

```
import threading
rlock = threading.RLock()
rlock.acquire()
rlock.acquire()                                     #程序不会堵塞
rlock.release()
rlock.release()
```

【例 19.6】 使用 lock 语句同步代码块示例（td_lock.py）。创建工作线程，模拟银行现金账户取款。当多个线程同时执行取款操作时，如果不使用同步处理（图 19-2(a)），会造成账户余额混乱；尝试使用同步锁对象 Lock，以保证多个线程同时执行取款操作时银行现金账户取款的有效和一致（图 19-2(b)）。

```
命令提示符
C:\pythonpa\ch19>python td_lock.py
Thread-4取款前余额: 200, 取款额: 81, 取款后额: 119
Thread-2取款前余额: 119, 取款额: 69, 取款后额: 50
Thread-1取款前余额: 50, 取款额: 70, 取款后额: -20
Thread-3取款前余额: -20, 取款额: 95, 取款后额: -115
Thread-5取款前余额: -115, 取款额: 71, 取款后额: -186
```

(a) 不使用同步锁，交易混乱

```
命令提示符
C:\pythonpa\ch19>python td_lock.py
Thread-1取款前余额: 200, 取款额: 67, 取款后额: 133
Thread-2取款前余额: 133, 取款额: 85, 取款后额: 48
Thread-3交易失败。取款前余额: 48, 取款额: 90
Thread-4交易失败。取款前余额: 48, 取款额: 78
Thread-5交易失败。取款前余额: 48, 取款额: 61
```

(b) 使用同步锁，交易正常

图 19-2　同步锁运行效果

```
import threading, time, random
class Account(threading.Thread):                         #继承 threading.Thread
    lock = threading.Lock()                              #创建锁
    def __init__(self, amount):                          #构造函数
        threading.Thread.__init__(self)                  #调用父类构造函数
        Account.amount = amount                          #账户金额
    def run(self):                                       #定义 run()方法
        self.withdraw()                                  #取款
    def withdraw(self):
        Account.lock.acquire()                           #获取锁
        t = threading.current_thread()
        a = random.choice(range(50,101))
        if Account.amount < a:
            print('{0}交易失败。取款前余额:{1},取款额:{2}'.format(t.name, Account.amount, a))
            Account.lock.release()
            return 0                                     #拒绝交易
        time.sleep(random.choice(range(5)))              #随机睡眠[0-5)秒
        prev = Account.amount
        Account.amount -= a                              #取款
        print('{0}取款前余额:{1}, 取款额:{2}, 取款后额:{3}'.format(t.name, prev, a,
Account.amount))
        Account.lock.release()                           #释放锁
def test():
    for i in range(5):                                   #创建 5 个线程对象并启动
        Account(200).start()
if __name__ == '__main__':
    test()
```

19.2.8 基于条件变量的同步和通信

在使用 Lock 对象 lock 同步代码块时,假设线程 A 获得同步锁 lock,在执行同步代码时需要等待某资源,而该资源由线程 B 提供。此时,线程 B 无法获取线程 A 占用的同步锁 lock,故无法提供线程 A 需要的资源,因而会导致死锁。

为了解决死锁的问题,可以使用基于条件变量(Condition)的线程间通信的通信机制 wait()、notify()和 notifyAll()。

条件变量(Condition)对象关联一个锁 lock。在创建 Condition 对象时,可以传入一个参数 lock,如果没有传入该参数,则会自动创建一个。

```
cv = threading.Condition(lock = None)
```

Condition 对象 cv 的 acquire()和 release()调用其关联的锁 lock,cv 还支持 with 语句:

```
with cv:
    同步操作
```

Condition 对象 cv 还包括对象方法 wait()、notify()和 notifyAll()。wait()/wait(timeout)释放锁 lock,并阻塞当前线程允许,直到其他线程使用 notify()和 notifyAll()唤醒后重新获得锁 lock。notify()唤醒一个等待线程,notifyAll()唤醒所有等待线程。

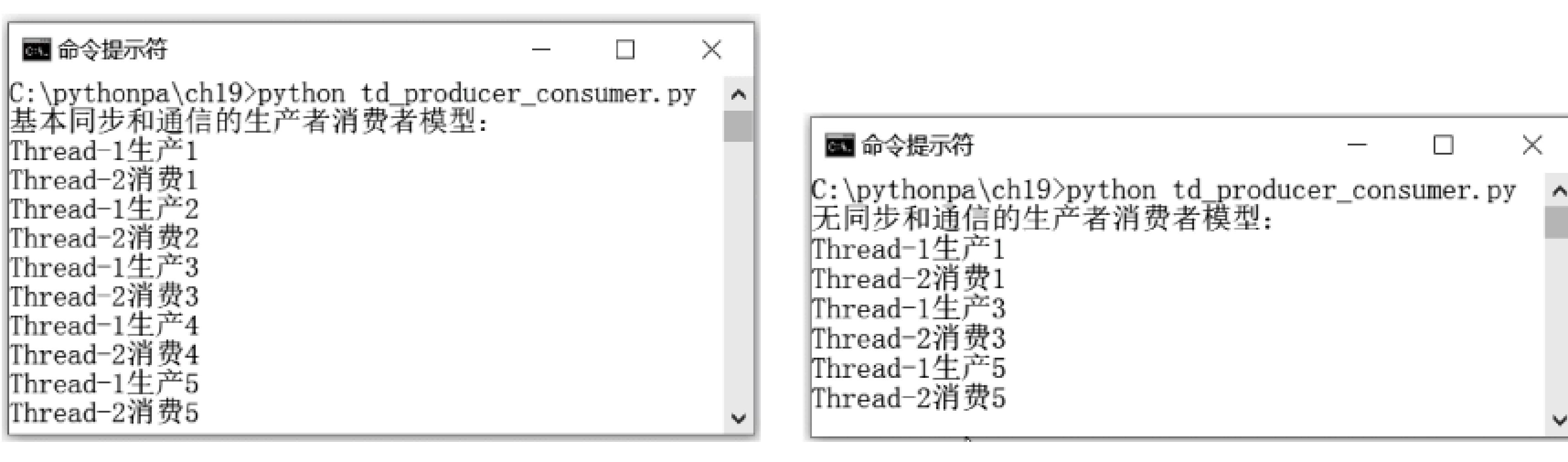
使用 Condition 对象 cv 进行线程通信避免死锁的原理可以简单概述为:线程 A 获得同步锁 lock,在执行同步代码时,如果需要等待线程 B 占用的某资源,则可以调用 wait()/wait(毫秒数)方法阻塞当前线程的运行,并释放其占用的同步锁 lock;然后调用 notify()方法,通知等待同步锁 lock 的线程 B。线程 B 获得同步锁 lock 后执行操作,释放 A 所需要的资源,然后释

放同步锁，并调用 notify()方法，通知等待同步锁 lock 的线程 A。线程 A 获得同步锁 lock，继续执行。

典型的生产者/消费者模型可以使用线程间通信，以保证生产者线程生产一件，消费者线程消费一件，二者保持同步。其基本代码片段如下：

```
#生产者代码片段
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()
#消费者代码片段
with cv:
    make_an_item_available()
    cv.notify()
```

【例 19.7】 线程间通信示例(td_producer_consumer.py)。生产者/消费者模型，使用线程间通信，生产者生产一件，消费者消费一件，二者保持同步(图 19-3(a))。若未使用线程同步(把最后一行代码改为 test2())，则结果无法预料(图 19-3(b))。



(a) 使用线程间通信，生产消费保持同步　　(b) 未使用线程同步，结果无法预料

图 19-3　线程间通信运行效果

```
import threading, time, random
class Container1():                              #基于同步和通信
    def __init__(self):                          #构造函数
        self.contents = 0                        #容器内容
        self.available = False                   #容器内容
        self.cv = threading.Condition()          #条件变量
    def put(self, value):                        #生产函数
        with self.cv:                            #使用条件变量同步
            if self.available:                   #如果已经生产,则等待
                self.cv.wait()                   #等待
            self.contents = value                #生产,设置内容
            t = threading.current_thread()
            print('{0}生产{1}'.format(t.name, self.contents))
            self.available = True                #设置容器状态:已生产
            self.cv.notify()                     #通知等待的消费者
    def get(self):                               #消费函数
        with self.cv:                            #使用条件变量同步
            if not self.available:               #如果已经生产,则等待
                self.cv.wait()                   #等待
            t = threading.current_thread()
            print('{0}消费{1}'.format(t.name, self.contents))
            self.available = False               #设置容器状态:未生产
            self.cv.notify()                     #通知等待的生产者
class Container2():                              #无同步和通信
```

```
    def __init__(self):                              #构造函数
        self.contents = 0                            #容器内容
        self.available = False                       #容器内容
    def put(self, value):                            #生产函数
        if self.available:                           #如果已经生产
            pass
        else:
            self.contents = value                    #生产,设置内容
            t = threading.current_thread()
            print('{0}生产{1}'.format(t.name, self.contents))
            self.available = True                    #设置容器状态:已生产
    def get(self):                                   #消费函数
        if not self.available:                       #如果已经生产,则等待
            pass
        else:
            t = threading.current_thread()
            print('{0}消费{1}'.format(t.name, self.contents))
            self.available = False                   #设置容器状态:未生产
class Producer(threading.Thread):                    #生产者类
    def __init__(self, container):                   #构造函数
        threading.Thread.__init__(self)              #调用父类构造函数
        self.container = container                   #容器
    def run(self):                                   #定义 run()方法
        for i in range(1,6):
            time.sleep(random.choice(range(5)))      #随机睡眠[0-5)秒
            self.container.put(i)                    #生产
class Consumer(threading.Thread):                    #消费者类
    def __init__(self, container):                   #构造函数
        threading.Thread.__init__(self)              #调用父类构造函数
        self.container = container                   #容器
    def run(self):                                   #定义 run()方法
        for i in range(1,6):
            time.sleep(random.choice(range(5)))      #随机睡眠[0-5)秒
            self.container.get()                     #消费
def test1():
    print('基本同步和通信的生产者消费者模型:')
    container = Container1()                         #创建容器
    Producer(container).start()                      #创建消费者线程并启动
    Consumer(container).start()                      #创建消费者线程并启动
def test2():
    print('无同步和通信的生产者消费者模型:')
    container = Container2()                         #创建容器
    Producer(container).start()                      #创建消费者线程并启动
    Consumer(container).start()                      #创建消费者线程并启动
if __name__ == '__main__':
    test1()
```

19.2.9 基于 queue 模块中队列的同步

Python 标准模块 queue 提供了适用于多线程编程的先进先出的数据结构(即队列),用于生产者和消费者线程之间的信息传递,使用 queue 模块中的线程安全的队列可以快捷地实现生产者和消费者模型。

在 queue 模块中包含 3 种线程安全的队列,即 Queue、LifoQueue 和 PriorityQueue。这里以 Queue 为例,其主要方法如下。

（1）Queue(maxsize=0)：构造函数，构造指定大小的队列。默认不限定大小。

（2）put(item, block=True, timeout=None)：向队列中添加一个项。默认阻塞，即队列满的时候程序阻塞等待。

（3）get(block=True, timeout=None)：从队列中拿出一个项。默认阻塞，即队列为空的时候程序阻塞等待。

【例 19.8】　基于 queue.Queue()的生产者和消费者模型(td_producer_consumer_queue.py)。

```
import time
import queue
import threading
q = queue.Queue(10)                                  #创建一个大小为 10 的队列
def productor(i):
    while True:
        time.sleep(1)                                #休眠 1 秒钟,即每秒钟做一个包子
        q.put("厨师{}做的包子!".format(i))            #如果队列满,则等待
def consumer(j):
    while True:
        print("顾客{}吃了一个{}" .format(j, q.get())) #如果队列空,则等待
        time.sleep(1)                                #休眠 1 秒钟,即每秒钟吃一个包子
for i in range(3):                                   #3 个厨师不停做包子
    t = threading.Thread(target = productor, args = (i,))
    t.start()
for k in range(10):                                  #10 个顾客等待吃包子
    v = threading.Thread(target = consumer, args = (k,))
    v.start()
```

程序运行结果如下。

```
顾客 9 吃了一个厨师 0 做的包子!
顾客 9 吃了一个厨师 1 做的包子!
顾客 9 吃了一个厨师 2 做的包子!
...
```

19.2.10　基于 Event 的同步和通信

threading.Event 是线程之间的通信机制之一：Event 对象管理一个标志(flag)，默认为 False。

Event 相当于红绿灯信号，可用于主线程控制其他线程的执行。当 flag 为 False 时，其他的线程调用 e.wait()阻塞等待这个信号；当设置 flag 为 True 时，等待的线程解除阻塞继续执行。

Event 对象主要包括下列方法。

（1）wait([timeout])：阻塞等待，直到 Event 对象的 flag 为 True 或超时。

（2）set()：将 flag 设置为 True。

（3）clear()：将 flag 设置为 False。

（4）isSet()：判断 flag 是否为 True。

【例 19.9】　基于 Event 的线程通信(td_event.py)。

```
import threading
import random
def f(i, e):
    e.wait()                                         #检测 Event 的标志,如果是 False 则阻塞
    print("线程{}的随机结果为{}".format(i, random.randrange(1,100)))
```

```
if __name__ == '__main__':
    event = threading.Event()                          #创建事件对象,默认标志为 False
    for i in range(3):                                 #创建 3 个线程并运行,默认阻塞等待 Event
        t = threading.Thread(target = f, args = (i, event))
        t.start()
    ready = input('请输入 1 开始继续执行阻塞的线程:')
    if ready == "1":
        event.set()                                    #设置 Event 的 flag 为 True
```

程序运行结果如下。

```
请输入 1 开始继续执行阻塞的线程:1
线程 0 的随机结果为 95
线程 2 的随机结果为 39
线程 1 的随机结果为 88
```

19.3 基于进程的并行计算

19.3.1 multiprocessing 模块概述

由于 GIL（全局解释锁）的问题，CPython 实现中的多线程不能充分利用多核 CPU 的处理资源。解决方法之一是使用 multiprocessing 模块基于进程进行并行计算。Python 标准库模块 multiprocessing 支持创建进程来实现并行计算，每个进程赋予单独的 Python 解释器，从而规避了 GIL 问题。

Python 标准库模块 multiprocessing 提供了与进程相关的操作，例如创建进程、启动进程、进程同步等。multiprocessing 模块还提供了进程池和线程池。

另外，multiprocessing 模块的 API 与 threading. Thread 类似，这大大减少了其使用的复杂度。

值得注意的是，创建进程与操作系统有关，UNIX/Linux 支持 fork 机制，而 Windows 平台只支持 spawn 机制，故在 Windows 平台中所有与进程相关的代码必须放置在“if __name__ == '__main__'：”之中，而且程序的调试运行需要在 Windows 命令行窗口使用 Python 程序名.py 的方式启动运行。

19.3.2 创建和使用进程

与创建线程 API 类似，创建进程也有两种方法，即创建 Process 的实例对象、创建 Process 的子类。下面以通过创建 Process 的实例对象来创建进程为例说明进程的创建和使用。创建进程的语法格式为：

```
Process(target = None, name = None, args = (), kwargs = {})   #构造函数
```

其中，target 是进程运行的函数；name 是进程的名称；args 和 kwargs 是传递给 target 的参数元组和命名参数字典。

通过调用 Process 对象的 start()方法可以启动进程。进程对象的常用方法如下。

- t. start()：启动线程。
- t. is_alive()：判断线程是否活动。
- t. join()：进程加入。
- terminate()：终止进程。

- t. name：进程名。
- t. pid：返回进程 ID。
- t. daemon：设置进程为用户进程(False)或 Daemon 进程(True)。

multiprocessing 模块包含了以下若干实用函数。

- cpu_count()：可用的 CPU 核数量。
- current_process()：返回当前进程。
- active_children()：活动的子进程。
- log_to_stderr()：函数可设置输出日志信息到标准错误输出(默认为控制台)。

【例 19.10】 使用 Process 对象创建和启动新进程(mp_process. py)。

```
import time, random
import multiprocessing as mp
def timer(interval):
    for i in range(3):
        time.sleep(random.choice(range(interval)))      #随机睡眠 interval 秒
        pid = mp.current_process().pid                  #获取当前进程 ID
        print('Process:{0} Time:{1}'.format(pid, time.ctime()))
if __name__ == '__main__':
    p1 = mp.Process(target = timer, args = (5,))        #创建进程
    p2 = mp.Process(target = timer, args = (5,))        #创建进程
    p1.start(); p2.start()                              #启动线程
    p1.join(); p2.join()
```

程序运行结果如下。

```
Process:5728 Time:Wed Jul 11 20:13:33 2018
Process:2192 Time:Wed Jul 11 20:13:34 2018
Process:2192 Time:Wed Jul 11 20:13:34 2018
Process:2192 Time:Wed Jul 11 20:13:34 2018
Process:5728 Time:Wed Jul 11 20:13:35 2018
Process:5728 Time:Wed Jul 11 20:13:38 2018
```

19.3.3　进程的数据共享

multiprocessing 模块为进程间通信提供了两种方法，即 Queue()和 Pipe()。

multiprocessing 模块中的 Queue()类似于 queue. Queue()(参见 19.2.9 节)，为进程间通信提供了一个线程和进程安全的队列。

multiprocessing 模块中的 Pipe()返回一个管道(包括两个连接对象)，两个进程可以分别连接到不同的端的连接对象，然后通过其 send()方法发送数据或者通过 recv()方法接收数据。

【例 19.11】 基于进程和 multiprocessing 模块中 Queue 队列的生产者和消费者模型(mp_queue. py)。

```
import time
import multiprocessing as mp
def productor(i, q):
    while True:
        time.sleep(1)                                   #休眠 1 秒钟，即每秒钟做一个包子
        q.put("厨师{}做的包子!".format(i))              #如果队列满，则等待
def consumer(j, q):
    while True:
        print("顾客{}吃了一个{}" .format(j, q.get())) #如果队列空，则等待
```

```
        time.sleep(1)                                # 休眠 1 秒钟,即每秒钟吃一个包子
if __name__ == '__main__':
    q = mp.Queue(10)                                 # 创建一个大小为 10 的队列
    for i in range(3):                               # 3 个厨师不停做包子
        p = mp.Process(target = productor, args = (i,q))
        p.start()
    for k in range(10):                              # 10 个顾客等待吃包子
        p = mp.Process(target = consumer, args = (k,q))
        p.start()
```

程序运行结果如下。

```
顾客 2 吃了一个厨师 1 做的包子!
顾客 3 吃了一个厨师 0 做的包子!
…
```

【例 19.12】 基于 multiprocessing 模块中 Pipe()的进程间通信(mp_pipe.py)。

```
import multiprocessing as mp
import time, random, itertools
def consumer(conn):
    # 从管道读取数据
    while True:
        try:
            item = conn.recv()
            time.sleep(random.randrange(2))          # 随机休眠,代表处理过程
            print("consume:{}".format(item))
        except EOFError:
            break
def producer(conn):
    # 生产项目并将其发送到连接的管道上
    for i in itertools.count(1):                     # 从 1 开始无限循环
        time.sleep(random.randrange(2))              # 随机休眠,代表处理过程
        conn.send(i)
        print("produce:{}".format(i))
if __name__ == "__main__":
    # 创建管道,返回两个连接对象的元组
    conn_out, conn_in = mp.Pipe()
    # 创建并启动生产者进程,传入参数管道一端的连接对象
    p_producer = mp.Process(target = producer, args = (conn_out,))
    p_producer.start()
    # 创建并启动消费者进程,传入参数管道另一端的连接对象
    p_consumer = mp.Process(target = consumer, args = (conn_in,))
    p_consumer.start()
    # 加入进程,等待完成
    p_producer.join(); p_consumer.join()
```

程序运行结果如下。

```
produce:1
produce:2
consume:1
produce:3
…
```

19.3.4　进程池

在使用多进程并行处理任务时,由于切换进程需要切换上下文环境,所以会造成 CPU 的大量开销。为解决这个问题,进程池的概念被提出。预先创建好一个较为优化的数量的进程

(一般为 CPU 的核数量)，形成进程池，然后按需把任务分配到不同进程中执行。

使用 Python 标准库模块 multiprocessing 中的 Pool 类可创建进程池。其大致步骤如下：

(1) 使用构造函数 Pool(processes, initializer, initargs)创建一个进程池对象。这 3 个参数均为可选参数。其中，processes 为进程池的进程数量，默认为 CPU 的核数量；initializer 和 initargs 为启动任务进程时执行的初始化函数及其参数。

(2) 调用进程池对象的方法执行任务，将返回结果收集为一个列表，包括 apply_async()、apply()、map_async()、map()等。其中，apply_async()和 map_async()是异步非阻塞模式，即启动进程函数之后会继续执行后续的代码，不用等待进程函数返回。进程池的 map()方法与内置的 map()函数一样，把函数应用于可迭代对象的每一个元素。

(3) 等待任务进程完成。调用 join()方法加入进程池，等待其完成。用户也可以调用 close()关闭进程池，不再加入新的任务(Pool 对象支持 with 上下文操作，自动调用 close()方法)；或者调用 terminate()直接终止进程池。

说明：multiprocessing.dummy.Pool 提供了进程池的实现。

【例 19.13】 进程池的使用(mp_pool.py)。

```
from multiprocessing import Pool, TimeoutError
import time
import os
def f(x):
    return x * x                                    #返回 x 的平方
if __name__ == '__main__':
    #创建 4 个进程的进程池,并且调用其对象方法并行执行各任务
    with Pool(processes = 4) as pool:
        #使用进程池对象的 map()函数,并行计算并返回结果
        res1 = pool.map(f, range(10))
        print("pool.map 的结果:{}".format(res1))
        #使用进程池对象的 apply_async()函数,异步执行一次任务
        res2 = pool.apply_async(f, (20,))           #异步求解 f(20),仅使用一个进程
        print(res2.get(timeout = 1))                #输出结果:400
        res3 = pool.apply_async(os.getpid, ())      #异步执行 os.getpid(),仅使用一个进程
        print(res3.get(timeout = 1))                #输出执行任务的进程的 PID
        res4 = pool.apply_async(time.sleep, (10,))  #异步睡眠 10 秒钟
        try:
            print(res4.get(timeout = 1))            #尝试获得结果,等待超时为 1 秒钟
        except TimeoutError:
            print("结果超时!")
        #使用列表解析式,可能使用多个进程
        res5 = [pool.apply_async(os.getpid, ()) for i in range(5)]
        print([res.get(timeout = 1) for res in res5])
        print("在 With 语句中,进程池可用")
    print("在 With 语句之外,进程池自动关闭,不再可用")
```

程序运行结果如下。

```
pool.map 的结果:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
400
13012
结果超时!
[13012, 10404, 16132, 13012, 16132]
在 With 语句中,进程池可用
在 With 语句之外,进程池自动关闭,不再可用
```

19.4 基于线程池/进程池的并发和并行任务

19.4.1 concurrent.futures 模块概述

当项目达到一定规模时,使用 threading 和 multiprocessing 模块频繁创建(销毁线)线程(进程)会耗费大量资源,解决方案之一就是创建线程池(进程池),以空间换时间。从 Python 3.2 开始,标准库提供了 concurrent.futures 模块实现了对 threading 和 multiprocessing 的进一步抽象,提供了编写线程池(进程池)的支持。包 concurrent 意指并发,而 futures 意指将在未来完成的操作。

concurrent.futures 模块包含两个主要类及其实用函数。

(1) Executor:表示任务执行器。它是抽象类,可以使用其子类 ThreadPoolExecutor(线程池任务执行器)或者 ProcessPoolExecutor(进程池任务执行器)创建任务执行器对象,其主要方法如下。

- submit(fn, * args, ** kwargs):调度一个执行 fn(* args ** kwargs)的任务,并返回 Future 对象。
- map(func, * iterables):调度执行多个执行 func()的任务。它类似于 map(func, * iterables),但任务模式为异步并行处理。
- shutdown(wait=True):关闭任务执行器,等待调度的任务完成,不再调度新的任务。注意:Executor 支持 with 上下文,自动调用 shutdown()方法。

(2) Future:表示将执行的任务。其主要方法如下。

- cancel():尝试取消任务的执行。
- done():如果任务完成或被取消,则返回 True。
- result(timeout=None):返回任务执行的结果。
- add_done_callback(fn):添加任务完成回调函数,任务完成或被取消后执行 fn(任务对象)。

(3) concurrent.futures 模块还包含下列实用函数。

- wait(fs, timeout=None, return_when=ALL_COMPLETED):等待所有的任务完成。
- as_completed(fs, timeout=None):返回完成的任务(Future 对象)的迭代器,可以循环获取完成任务的结果。

19.4.2 使用 ThreadPoolExecutor 并发执行任务

ThreadPoolExecutor(线程池任务执行器)是 Executor 的派生类,用于使用一个线程池异步执行任务。由于 CPython 的 GIL 限制,ThreadPoolExecutor 适用于 IO 密集的任务。在创建 ThreadPoolExecutor 对象时可以指定其线程数:

```
ThreadPoolExecutor(max_workers = None, thread_name_prefix = '')
```

【例 19.14】 使用 ThreadPoolExecutor 并发爬取网页(future_tpe_get_pages.py)。

```
import concurrent.futures as cf
import time, urllib.request
```

```
def load_page(url):
    with urllib.request.urlopen(url, timeout = 60) as conn:
        return ('{}主页大小:{}字节'.format(url, len(conn.read())))
if __name__ == '__main__':
    URLS = ['http://www.163.com', 'http://www.sina.com.cn/', 'http://www.sohu.com/']
    #传统串行方法
    start_time = time.time()
    for url in URLS:
        print(load_page(url))
    end_time = time.time()
    print("串行处理消耗时间:{}".format(end_time - start_time))
    #使用 ThreadPoolExecutor 并发处理
    start_time = time.time()
    executor = cf.ThreadPoolExecutor()
    wait_for = [executor.submit(load_page, url) for url in URLS]
    for f in cf.as_completed(wait_for):                #迭代完成的任务,输出其结果
        print(f.result())
    end_time = time.time()
    print("并发处理消耗时间:{}".format(end_time - start_time))
```

程序运行结果如下。

```
http://www.163.com 主页大小:701216 字节
http://www.sina.com.cn/主页大小:570385 字节
http://www.sohu.com/主页大小:207315 字节
串行处理消耗时间:2.130052328109741
http://www.sina.com.cn/主页大小:570385 字节
http://www.163.com 主页大小:701216 字节
http://www.sohu.com/主页大小:207315 字节
并发处理消耗时间:1.5238358974456787
```

19.4.3　使用 ProcessPoolExecutor 并发执行任务

ProcessPoolExecutor(进程池任务执行器)是 Executor 的派生类,用于使用一个进程池异步执行任务。ProcessPoolExecutor 基于 multiprocessing 模块,因而避免了 CPython 的 GIL 限制,从而适用于计算密集的任务。在创建 ProcessPoolExecutor 对象时可以指定其进程数:

```
ProcessPoolExecutor(max_workers = None)
```

【例 19.15】 使用 ProcessPoolExecutor 求解最大公约数(future_ppe_gcd.py)。

```
import time
import concurrent.futures as cf
def gcd(pair):
    #求最大公约数
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
if __name__ == '__main__':
    #测试数据
    TEST_DATA = [(11880774, 83664910), (13961044, 17644234), (10112000, 13380625)]
    #传统串行方法
    start_time = time.time()
    res1 = list(map(gcd, TEST_DATA))
    end_time = time.time()
```

```
print("串行处理结果:{},消耗时间:{}".format(res1, end_time - start_time))
#使用 ProcessPoolExecutor 并行处理
start_time = time.time()
pool = cf.ProcessPoolExecutor(max_workers=4)
res2 = list(pool.map(gcd, TEST_DATA))
end_time = time.time()
print("并行处理结果:{},消耗时间:{}".format(res2, end_time - start_time))
```

程序运行结果如下。

```
串行处理结果:[35678, 35078, 9875],消耗时间:2.4343101978302
并行处理结果:[35678, 35078, 9875],消耗时间:1.5229299068450928
```

19.5 基于 asyncio 的异步 IO 编程

19.5.1 asyncio 模块概述

异步 IO(Asynchronous IO)是指程序发起一个 IO 操作(阻塞等待)后不用等 IO 操作结束可以继续其他操作,做其他事情,当 IO 操作结束时会得到通知,然后继续执行。异步 IO 编程是实现并发的一种方式,适用于 IO 密集型任务。

Python 标准库模块 asyncio 提供了一个异步编程框架,主要包括下列部分:

1. 事件循环(event loop)

基于 asyncio 的异步编程框架的核心是事件循环对象。使用 asyncio.get_event_loop()可以直接返回默认的事件循环对象,也可以选择创建其他事件循环对象。

事件循环对象用于创建任务(基于协程)和调度任务,用于高效地处理 IO 事件、系统事件、应用环境切换等。

2. 协程(coroutine)

使用 async 关键字定义的函数称为异步函数,其调用不会立即执行函数,而是会返回一个协程对象。协程对象需要封装为任务,注册到事件循环中,由事件循环调度。

3. 任务(Task)和 Future(将执行的任务)

协程对象不能直接运行,而是需要封装为任务,然后通过事件循环对象来调度运行。Task 是 Future 类的子类,任务(Task)是对协程的进一步封装,其中包含任务的各种状态。Future 对象代表将来执行或没有执行的任务的状态和结果。

19.5.2 创建协程对象

通过 async 关键字定义一个异步函数,调用异步函数返回一个协程(coroutine)对象。协程也是一种对象,协程不能直接运行,需要把协程加入到事件循环中,由后者在适当的时候调用。

使用 asyncio.get_event_loop()方法可以创建一个事件循环对象,然后使用其 run_until_complete()方法将协程注册到事件循环。

在异步函数中,可以使用 await 关键字针对耗时的操作(例如网络请求、文件读取等 IO 操作)进行挂起。当协程执行到 await 语句时,事件循环将会挂起该协程(函数让出控制权),执行其他协程。这和生成器里的 yield 一样,事实上,Python 早期版本使用 yield from 实现协程。另外,用户可以使用 asyncio.sleep()函数(休眠一段时间)来模拟 IO 操作。

【例 19.16】　创建协程(coroutine)对象示例(async_coroutine.py)。

```
import asyncio, time
async def do_some_work(n):                          # 使用 async 关键字定义异步函数
    print('等待:{}秒'.format(n))
    await asyncio.sleep(n)                          # 休眠一段时间
    return '{}秒后返回结束运行'.format(n)
start_time = time.time()                            # 开始时间
coro = do_some_work(2)
loop = asyncio.get_event_loop()
loop.run_until_complete(coro)
print('运行时间: ', time.time() - start_time)
```

程序运行结果如下。

```
等待:2 秒
运行时间: 2.010594367980957
```

19.5.3　创建任务对象

任务(Task)对象用于封装协程对象,保存了协程运行后的状态,用于未来获取协程的结果。

用户可以使用 asyncio.ensure_future(coroutine)创建一个任务对象,也可以使用事件循环对象的 create_task(coroutine)方法创建任务。

使用 run_until_complete()方法将任务注册到事件循环,同时注册多个任务的列表可以使用 run_until_complete(asyncio.wait(tasks)),注册多个任务可以使用 run_until_complete(asyncio.gather(*tasks))。

调用任务对象的 cancel()方法可以取消任务,result()方法返回结果。

【例 19.17】　创建任务对象示例(async_task.py)。

```
import asyncio, time
async def do_some_work(i, n):                       # 使用 async 关键字定义异步函数
    print('任务{}等待: {}秒'.format(i, n))
    await asyncio.sleep(n)                          # 休眠一段时间
    return '任务{}在{}秒后返回结束运行'.format(i, n)
start_time = time.time()                            # 开始时间
tasks = [asyncio.ensure_future(do_some_work(1, 2)),
         asyncio.ensure_future(do_some_work(2, 1)),
         asyncio.ensure_future(do_some_work(3, 3))]
loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
for task in tasks:
    print('任务执行结果: ', task.result())
print('运行时间: ', time.time() - start_time)
```

程序运行结果如下(运行结果表明,并发总运行时间 3s 小于所需运行之和 6s):

```
任务 1 等待: 2 秒
任务 2 等待: 1 秒
任务 3 等待: 3 秒
任务执行结果: 任务 1 在 2 秒后返回结束运行
任务执行结果: 任务 2 在 1 秒后返回结束运行
任务执行结果: 任务 3 在 3 秒后返回结束运行
运行时间: 3.0485544204711914
```

19.6 应用举例

19.6.1 使用 Pool 并行计算查找素数

下面以查找小于 n 的所有素数为例比较常规的串行处理(结果写入 prime1.txt)和基于进程池 Pool 的并行处理(结果写入 prime2.txt)的时间消耗。

【例 19.18】 并行查找小于 n 的所有素数(np_prime.py)。

```
import math
import time
import multiprocessing
def isprime(n):
    """判断 n 是否为素数,如果是,返回 n,否则返回 0"""
    if n < 2:
        return 0
    if n == 2:
        return n
    k = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= k:
        if n % i == 0:
            return 0
        i += 1
    return n
if __name__ == "__main__":
    #测试数据
    test_data = range(10 ** 6)
    #串行处理测试
    start_time = time.time()                        #结束时间
    with open("prime1.txt", "w") as outf:
        for num in test_data:
            r = isprime(num)
            if r > 0:
                outf.writelines("{}\n".format(num))
    end_time = time.time()
    print("串行处理消耗时间:{}".format(end_time - start_time))
    #并行处理测试
    start_time = time.time()                        #开始时间
    pool = multiprocessing.Pool(4)
    resultList = pool.map(isprime, test_data)
    pool.close()
    pool.join()
    with open("prime2.txt", "w") as outf:
        for r in resultList:
            if r > 0:
                outf.writelines("{}\n".format(r))
    end_time = time.time()                          #结束时间
    print("并行处理消耗时间:{}".format(end_time - start_time))
```

程序运行结果如下(两种方法查找的素数分别写入 prime1.txt 和 prime2.txt 中)。

```
串行处理消耗时间:8.06755781173706
并行处理消耗时间:5.192432165145874
```



19.6.2　使用 ProcessPoolExecutor 并行判断素数

下面以判断超大整数是否为素数为例比较常规的串行处理和基于 ProcessPoolExecutor 的并行处理的时间消耗。

【例 19.19】 使用 ProcessPoolExecutor 并行判断素数(future_ppe_prime.py)。

```
import concurrent.futures as cf
import math, time
def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True
if __name__ == "__main__":
    #测试数据
    test_data = [112272535095293, 112272535095293, 115280095190773, 1099726899285419]
    #串行处理测试
    start_time = time.time()                    #结束时间
    for num in test_data:
        print('{}是素数否:{}'.format(num, is_prime(num)))
    end_time = time.time()
    print("串行处理消耗时间:{}".format(end_time - start_time))
    #并行处理测试
    start_time = time.time()                    #开始时间
    with cf.ProcessPoolExecutor() as executor:
        primes = executor.map(is_prime, test_data)
        for number, prime in zip(test_data, primes):
            print('{}是素数否:{}'.format(number, prime))
    end_time = time.time()
    print("并行处理消耗时间:{}".format(end_time - start_time))
```

程序运行结果如下。

```
112272535095293 是素数否:True
112272535095293 是素数否:True
115280095190773 是素数否:True
1099726899285419 是素数否:False
串行处理消耗时间:2.1492953300476074
112272535095293 是素数否:True
112272535095293 是素数否:True
115280095190773 是素数否:True
1099726899285419 是素数否:False
并行处理消耗时间:1.2875590324401855
```

19.6.3　使用 ThreadPoolExecutor 多线程爬取网页

下面以批量下载网页内容为例比较常规的串行处理和基于 ThreadPoolExecutor 的并发处理的时间消耗。

【例 19.20】 使用 ThreadPoolExecutor 批量下载网页内容(future_tpe_download.py)。

```
import concurrent.futures
import urllib.request
import time
def load_url(url, timeout):
    """读取指定 URL 的网页数据"""
    with urllib.request.urlopen(url, timeout = timeout) as conn:
        return conn.read()
if __name__ == '__main__':
    #测试数据,GutenBerg 网站 TOP 9 Ebooks
    URLS = {'Pride and Prejudice': 'http://www.gutenberg.org/files/1342/1342-0.txt',
            'Heart of Darkness': 'http://www.gutenberg.org/files/219/219-0.txt',
            'Moby Dick': 'http://www.gutenberg.org/files/2701/2701-0.txt',
            'Frankenstein': 'http://www.gutenberg.org/files/84/84-0.txt',
            'Manual of Classical Erotology': 'http://www.gutenberg.org/files/57284/57284-0.txt',
            'A Tale of Two Cities': 'http://www.gutenberg.org/files/98/98-0.txt',
            'Alice Adventures in Wonderland': 'http://www.gutenberg.org/files/11/11-0.txt',
            'The Adventures of Tom Sawyer': 'http://www.gutenberg.org/files/74/74-0.txt',
            'Grimms Fairy Tales': 'http://www.gutenberg.org/files/2591/2591-0.txt'}
    #串行处理测试
    start_time = time.time()                              #结束时间
    for name, url in URLS.items():
        data = load_url(url, 60)
        with open("{}.txt".format(name), 'wb') as f:      #保存下载的文件
            f.write(data)
        print('下载{}完成,文件大小为{}字节'.format(name, len(data)))
    end_time = time.time()
    print("串行处理消耗时间:{}".format(end_time - start_time))
    #并行处理测试
    start_time = time.time()                              #开始时间
    #使用上下文创建线程池执行器,以确保其关闭
    with concurrent.futures.ThreadPoolExecutor(max_workers = 5) as executor:
        #使用 ThreadPoolExecutor 调度任务
        future_to_name = {executor.submit(load_url, url, 60): name for name, url in URLS.items()}
        #迭代已完成的任务,并输出结果
        for future in concurrent.futures.as_completed(future_to_name):
            name = future_to_name[future]
            try:
                data = future.result()
            except Exception as exc:
                print('下载{}时出错{}'.format(name, exc))
            else:
                with open("{}.txt".format(name), 'wb') as f:  #保存下载的文件
                    f.write(data)
                print('下载{}完成,文件大小为{}字节'.format(name, len(data)))
    end_time = time.time()
    print("并行处理消耗时间:{}".format(end_time - start_time))
```

程序运行结果如下(结果表明,并发处理对应 IO 密集处理,可以显著减少时间消耗)。

```
下载 Pride and Prejudice 完成,文件大小为 724725 字节
下载 Heart of Darkness 完成,文件大小为 234041 字节
下载 Moby Dick 完成,文件大小为 1270330 字节
下载 Frankenstein 完成,文件大小为 450783 字节
下载 Manual of Classical Erotology 完成,文件大小为 331400 字节
下载 A Tale of Two Cities 完成,文件大小为 804335 字节
```

```
下载 Alice Adventures in Wonderland 完成,文件大小为 173595 字节
下载 The Adventures of Tom Sawyer 完成,文件大小为 428104 字节
下载 Grimms Fairy Tales 完成,文件大小为 560166 字节
串行处理消耗时间:87.37381529808044
下载 Heart of Darkness 完成,文件大小为 234041 字节
下载 Manual of Classical Erotology 完成,文件大小为 331400 字节
下载 Alice Adventures in Wonderland 完成,文件大小为 173595 字节
下载 Pride and Prejudice 完成,文件大小为 724725 字节
下载 Moby Dick 完成,文件大小为 1270330 字节
下载 Grimms Fairy Tales 完成,文件大小为 560166 字节
下载 A Tale of Two Cities 完成,文件大小为 804335 字节
下载 The Adventures of Tom Sawyer 完成,文件大小为 428104 字节
下载 Frankenstein 完成,文件大小为 450783 字节
并行处理消耗时间:19.013251543045044
```

19.7　复　习　题

1. Python 可以使用________创建一个线程并运行指定函数，当函数返回时线程自动结束，也可以通过________结束线程。

2. Python 可通过声明 Thread 的派生类，并重写对象的________方法，然后创建其对象实例来创建线程；通过对象的________方法可以启动线程，并自动执行对象的 run()方法。

3. 线程可分为________和________。

4. ________又称守护线程，其优先级是最低的，一般为其他的线程提供服务。

5. Lock 对象锁有两个状态，即________和________。

19.8　上 机 实 践

完成本章中的例 19.1～例 19.20，熟悉 Python 语言并行计算(线程、进程和协程)程序设计。

19.9　案例研究：文本统计并行处理

大量的文本统计可通过并行处理来缩短处理时间。本章案例研究通过使用进程池遍历指定目录下的所有文本文件和扩展名为.py 的文件，统计其行数与字数，将结果写入到文本文件，并比较串行执行的时间消耗，帮助读者进一步深入了解 Python 并行计算的方法和流程。

本章案例研究的解题思路和源代码等以电子版形式提供，具体请扫描如下二维码。



案例研究

第 20 章 系统管理



视频讲解

Python 是一种非常适合系统管理员的脚本编写语言，使用 Python 可以实现各种复杂的系统管理工作。

20.1 系统管理相关模块

Python 标准库中包括下列与系统管理相关的模块。

- os 模块：与操作系统相关的函数。
- os.path 模块：与路径相关的函数。
- glob 模块：文件通配符操作。
- tempfile 模块：创建临时目录和文件。
- shutil 模块：与目录和文件操作相关的函数。
- subprocess 模块：用于执行其他程序。

20.2 目录、文件和磁盘的基本操作

20.2.1 创建目录

使用 os 模块中的下列函数可以创建目录，其语法形式如下：

- **os.mkdir(path, mode = 0o777)**　　　#创建目录 path
- **os.makedirs(path, mode = 0o777)**　　#创建目录 path，以及所有 path 中包含的上级目录

其中，path 为指定目录。如果 path 已存在，则将导致 FileExistsError。例如：

```
>>> import os
>>> os.makedirs(r'c:\pythonpa\ch20\temp\dir1')
>>> os.mkdir(r'c:\pythonpa\ch20\dir2')
```

20.2.2 临时目录和文件的创建

使用 tempfile 模块中的下列函数可以创建临时目录和文件，其语法形式如下：

- **tempfile.mkdtemp(suffix = '', prefix = 'tmp', dir = None)**　#创建并返回临时目录
- **tempfile.mkstemp(suffix = '', prefix = 'tmp', dir = None, text = False)**　#创建并返回临时文件
- **tempfile.TemporaryDirectory(suffix = '', prefix = 'tmp', dir = None)**　#调用 mkdtemp()，创建临时目录
- **tempfile.TemporaryFile(mode = 'w + b', buffering = None, encoding = None, newline = None, suffix = '', prefix = 'tmp', dir = None)**　#调用 mkstemp()，创建临时文件

- **tempfile.tempdir**　　　　　　　　　　　　　# 设置临时目录对应的路径
- **tempfile.gettempdir()**　　　　　　　　　　# 获取临时目录

临时目录和文件只在程序运行时有效，当文件关闭时系统会自动删除。例如：

```
>>> import tempfile
>>> tempfile.gettempdir()                #输出:'C:\\Users\\jh\\AppData\\Local\\Temp
>>> tempfile.mkstemp()
(3, 'C:\\Users\\jh\\AppData\\Local\\Temp\\tmpvl9ii99g')
>>> tempfile.mkdtemp()
'C:\\Users\\jh\\AppData\\Local\\Temp\\tmp3eoscx8h'
```

20.2.3　切换和获取当前工作目录

使用 os 模块中的 chdir()函数可以切换当前工作目录，其语法形式如下：

```
os.chdir(path)                           #切换当前工作目录为 path
```

其中，path 为指定文件。如果找不到 path，则将导致 FileNotFoundError。例如：

```
>>> os.chdir(r'c:\pythonpa')
```

使用 os 模块中的 getcwd()函数可以获取当前工作目录，其语法形式如下：

```
os.getcwd()                              #获取当前目录/路径
```

20.2.4　目录内容列表

使用 os 模块中的 listdir()函数可以显示一个目录中的文件/子目录列表，其语法形式如下：

```
os.listdir(path = '.')                   #返回指定目录 path 中所有文件/子目录的列表
```

其中，path 为指定目录，默认为当前目录'.'。os.curdir 也表示当前目录。例如：

```
>>> os.listdir(r'c:\pythonpa')
['ch01', 'ch02', 'ch03', 'ch04', 'ch06', 'ch07', 'ch08', 'ch09', 'ch10', 'ch11', 'ch12', 'ch13',
'ch14', 'ch15', 'ch17', 'ch18', 'ch19', 'ch20', 'images']
```

20.2.5　文件通配符和 glob.glob()函数

使用 glob 模块中的 glob()函数可以获取满足指定模式的文件/目录列表，其语法形式如下：

```
glob.glob(pathname)                      #返回满足指定模式 pathname 的文件/目录的列表
```

其中，pathname 为目录/文件模式，可以包含通配符 *（0 或多个字符）和?（1 个字符）。例如：

```
>>> import glob
>>> os.chdir(r'c:\pythonpa\ch01')
>>> glob.glob('*.py')
['bigint.py', 'hello.py', 'hello1.py', 'hello_argv.py']
```

20.2.6　遍历目录和 os.walk()函数

使用 os 模块中的 walk()函数可以遍历指定的目录结构，其语法形式如下：

```
os.walk(top, topdown = True, onerror = None, followlinks = False)   #返回目录结构的迭代器
```

其中，top 为起始目录；topdown 若为 False，则从下往上遍历。对于目录结构中的每一个目录，生成一个元组(dirpath, dirnames, filenames)，dirpath 为目录，dirnames 为其中包含的子

目录列表，filenames 为其中包含的文件列表。

使用 os 模块中的 join()函数可以将目录名和文件名连接成全限定路径，其语法形式如下：

```
os.path.join(path1[, path2[, …]])
```

【例 20.1】 输出指定目录的目录结构（oswalk.py）。

```
import re, os, os.path
def ls_py(top):
    for (dirname, subdirs, files) in os.walk(top):
        print('[' + dirname + ']')
        for fname in files:
            print(os.path.join(dirname, fname))
#测试代码
if __name__ == '__main__':
    path1 = r'c:\pythonpa\ch17'
    ls_py(path1)
```

程序运行结果如下。

```
[c:\pythonpa\ch17]
c:\pythonpa\ch17\DBCreate.py
c:\pythonpa\ch17\DBquery.py
c:\pythonpa\ch17\DBUpdate.py
c:\pythonpa\ch17\sales.db
```

20.2.7 判断文件/目录是否存在

使用 os.path 模块中的 exists()函数，可以判断文件/目录是否存在，其语法形式如下：

```
os.path.exists(路径名)
```

例如：

```
>>> import os.path
>>> os.path.exists(r'c:\abc')            #假设"c:\abc"不存在。输出:False
```

20.2.8 测试文件类型

文件名、目录名和链接名都是用一个字符串作为其标识符。使用 os.path 模块中的下列函数可以判断其类型，语法形式如下：

- **os.path.isfile(path)**　　#路径 path 是否为文件类型
- **os.path.isdir(path)**　　#路径 path 是否为目录类型
- **os.path.islink(path)**　　#路径 path 是否为链接类型
- **os.path.ismount(path)**　　#路径 path 是否为装载点类型
- **os.path.isabs(path)**　　#路径 path 是否为绝对路径

例如：

```
>>> os.path.isdir(r'c:\pythonpa')            #输出:True
```

20.2.9 文件的日期及大小

使用 os.path 模块中的下列函数可以获取文件和目录的其他属性，语法形式如下：

- **os.path.getatime(path)**　　#返回上次访问时间
- **os.path.getmtime(path)**　　#返回上次修改时间
- **os.path.getctime(path)**　　#返回创建时间
- **os.path.getsize(path)**　　#返回指定路径 path 的大小

其中，path 为指定文件目录路径，默认为当前目录'.'。例如：

```
>>> import os.path, time
>>> os.path.getctime(r'c:\pythonpa\ch01') #结果为秒.输出:1531306262.9783783
>>> time.strftime('%c',time.gmtime(os.path.getctime(r'c:\pythonpa\ch01')))
'Wed Jul 11 10:51:02 2018'
```

20.2.10　文件和目录的删除

1. 删除文件

使用 os 模块中的 remove()函数可以删除指定文件，其语法形式如下：

```
os.remove(path)                    #删除指定文件 path
```

其中，path 为指定文件。如果找不到 path，则将导致 FileNotFoundError；如果 path 为目录，则将导致 PermissionError。例如：

```
>>> os.remove(r'c:\pythonpa\temp\1.txt')
```

2. 删除目录

使用 os 模块中的 rmdir()函数可以删除指定目录，其语法形式如下：

```
os.rmdir(path)                     #删除指定目录 path
```

其中，path 为指定目录。如果找不到 path，则将导致 FileNotFoundError；如果目录不为空，则将导致 OSError。例如：

```
>>> os.rmdir(r'c:\pythonpa\temp')
```

使用 shutil 模块中的 rmtree()函数可以删除指定目录及目录下的所有内容，其语法形式如下：

```
shutil.rmtree(path)                #删除指定目录 path
```

20.2.11　文件和目录的复制、重命名和移动

使用 shutil 模块中的下列函数可以复制文件和目录，语法形式如下：

- **shutil.copy(src, dst)**　　#复制文件 src 到 dst，如果 dst 为目录，则复制到 dst 目录下
- **shutil.copy2(src, dst)**　　#复制文件 src 到 dst，如果 dst 为目录，则复制到 dst 目录下
- **shutil.copytree(src, dst, symlinks = False, ignore = None)**　#复制目录树 src 到 dst
- **shutil.move(src, dst)**　　#将文件/目录 src 移动到 dst

其中，src 为源路径；dst 为目标路径。copy()除了复制文件内容外，还复制文件许可权限；copy2()则复制所有元数据，包括创建时间和修改时间。例如：

```
>>> import shutil
>>> shutil.copytree(r'c:\pythonpa\ch20\temp', r'c:\pythonpa\ch20\temp1')
```

在复制目录树时，可以指定忽略的文件。通常使用 shutil.ignore_patterns(*patterns)返回的函数对象。例如：

```
>>> shutil.copytree(r'c:\pythonpa', r'c:\pythonbak', ignore = shutil.ignore_patterns('*~','*.pyc'))
```

20.2.12　磁盘的基本操作

使用 shutil 模块中的 disk_usage()函数可以获取磁盘空间的使用情况，其语法形式如下：

```
shutil.disk_usage(path)      #返回指定 path 上的磁盘的空间使用情况,形式为(总数, 已用, 可用)
```

例如：

```
>>> import shutil; shutil.disk_usage(r'c:')
usage(total=254721126400, used=236444225536, free=18276900864)
```

20.3 执行操作系统命令和运行其他程序

20.3.1 os.system()函数

使用 os 模块中的 system()函数可以在 Python 程序中执行操作系统的命令和脚本，或运行其他程序，语法形式如下：

```
os.system(command)              #执行操作系统命令,返回命令执行结果的返回代码
```

例如：

```
>>> os.system('dir')            #执行操作系统命令
>>> os.system('notepad.exe')    #执行程序,启动记事本
```

20.3.2 os.popen()函数

使用 os 模块中的 popen()函数可以在 Python 程序中执行操作系统的命令和脚本，语法形式如下：

```
os.popen(...)                   #执行操作系统命令,返回打开的管道(相当于文件)
```

例如：

```
>>> os.popen(r'dir c:\pythonpa')     #输出:<os._wrap_close object at 0x000001BB25F71048>
>>> list(os.popen(r'dir c:\pythonpa'))
['驱动器 C 中的卷是 Windows\n', '卷的序列号是 FA31-CCF8\n', '\n', 'c:\\pythonpa 的目录\n', '\n',
'2018/07/10  15:14    <DIR>          .\n', '2018/07/10  15:14    <DIR>          ..\n',
'2018/07/10  13:33    <DIR>          ch01\n', '2018/07/10  13:33    <DIR>          ch02\n',
'2018/07/10  13:33    <DIR>          ch03\n', '2018/07/10  13:33    <DIR>          ch04\n',
'2018/07/10  13:33    <DIR>  ...
```

20.3.3 subprocess 模块

subprocess 模块提供了若干函数和对象，用于创建子进程、运行外部程序、连接到其输入/输出/错误管道、获取其返回值。subprocess 模块用于取代 os.system()和 os.popen()函数，提供了更高级的功能。

subprocess 模块函数 call()、check_call()、check_output()用于执行外部程序。call()返回 returncode；如果 returncode 不为 0，则 check_call()引发 CalledProcessError；check_output()返回程序运行结果。它们的语法形式如下：

- **call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)**
- **check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)**
- **check_output(args, *, stdin=None, stderr=None, shell=False, universal_newlines=False, timeout=None)**

其中，args 是外部程序及其参数列表。若 shell 设定为 True，则执行操作系统命令。例如：

```
>>> import subprocess
>>> subprocess.call(['notepad.exe', r'c:\pythonpa\ch01\hello.py'])   #在记事本中打开指定文件
```

```
>>> subprocess.check_output(r'python -h', shell=True)
b"usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...\r\n...
```

如果需要与所创建的子进程进行高级通信，例如传递输入参数，可以使用 subprocess 模块的 Popen 对象构造函数，语法形式如下：

```
Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_
fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False,
startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=())
```

Popen 对象包含下列方法和属性：

```
• poll()                 #检查子进程是否终止
• wait(timeout=None)     #等待子进程终止
• communicate(input=None, timeout=None)     #发送数据给子进程
• send_signal(signal)    #发送信号给子进程
• terminate()            #终止子进程
• kill()                 #强行终止子进程
• stdin/stdout/stderr    #子进程的输入/输出/错误文件对象(构造函数 stdin/stdout/stderr 为
                         #subprocess.PIPE 时)
• pid                    #子进程的进程 ID(当 shell=True,即执行操作系统命令时,为 Shell 的 pid)
• returncode             #子进程的返回值
```

例如：

```
>>> import subprocess
>>> p = subprocess.Popen(['dir'], shell=True, stdout=subprocess.PIPE, stdin=subprocess
.PIPE)
>>> stdoutdata, stderrdata = p.communicate()
>>> stdoutdata
b'\xc7\xfd\xb6\xaf\xc6\xf7 C \xd6\xd0\xb5\xc4\xbe\xed\xca\xc7 Windows\r\n...
```

20.4　获取终端的大小

通过 os 或 shutil 模块的 get_terminal_size()函数可以获取终端的大小，以方便输出内容的格式化操作，语法形式如下：

```
• os.get_terminal_size(fd=STDOUT_FILENO)       #获取并返回终端大小
• shutil.get_terminal_size(fallback=(80, 24))  #获取并返回终端大小
```

两者返回 os.terminal_size 对象为元组的子类，包含(columns, lines)，以及窗口的列数和行数。通常建议使用高级别的函数 shutil.get_terminal_size()。

【例 20.2】 获取终端的大小示例(get_term_size.py)。

```
import os, shutil
def get_term_size_test():
    sz = shutil.get_terminal_size()
    print('窗口大小:', sz)
    for i in range(sz.lines):
        print('*' * sz.columns)
if __name__ == '__main__':
    get_term_size_test()
```

程序运行过程和结果如下。

```
c:\pythonpa\ch20>get_term_size.py
窗口大小: os.terminal_size(columns=80, lines=24)
********************************************************************************
...
```

20.5 文件的压缩和解压缩

Python 支持常用压缩格式(.tar、.tgz 和.zip)文件的压缩和解压缩功能。

使用 shutil 模块的 make_archive()和 unpack_archive()等函数可以实现文件的压缩和解压缩功能。shutil 模块实现高级别的操作,依赖于 zipfile 和 tarfile 模块。

20.5.1 shutil 模块支持的压缩和解压缩格式

shutil 模块的 get_archive_formats()和 get_unpack_formats()函数返回支持的压缩和解压缩格式。例如:

```
>>> import shutil
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"), ('tar', 'uncompressed tar file'),
('xztar', "xz'ed tar-file"), ('zip', 'ZIP file')]
>>> shutil.get_unpack_formats()
[('bztar', ['.tar.bz2', '.tbz2'], "bzip2'ed tar-file"), ('gztar', ['.tar.gz', '.tgz'], "gzip'ed tar
-file"), ('tar', ['.tar'], 'uncompressed tar file'), ('xztar', ['.tar.xz', '.txz'], "xz'ed tar-
file"), ('zip', ['.zip'], 'ZIP file')]
```

额外的压缩和解压缩格式可以使用 shutil 模块的注册和取消注册功能:

- **shutil.register_archive_format(name, function, extra_args=None, description='')**
- **shutil.unregister_archive_format(name)**
- **shutil.register_unpack_format(name, extensions, function, extra_args=None, description='')**
- **shutil.unregister_unpack_format(name)**

20.5.2 make_archive()函数和文件压缩

make_archive()函数的基本格式为:

make_archive(base_name, format, root_dir=None, base_dir=None, verbose=0, dry_run=0, owner=None, group=None, logger=None)

其中,base_name 是目标文件的路径,不包括文件扩展名;format 是文件格式,为'zip'、'tar'、'bztar'或'gztar';root_dir 是压缩文件的根目录,默认为当前目录;base_dir 是压缩的起始目录,默认为当前目录。例如:

```
>>> shutil.make_archive('pybak','zip', root_dir=r'c:\pythonpa', base_dir=r'c:\tmp')
'c:\\tmp\\pybak.zip'
```

20.5.3 unpack_archive()函数和文件解压缩

unpack_archive()函数的基本格式为:

unpack_archive(file_name, extract_dir=None, format=None)

其中,file_name 是压缩文件的名称;extract_dir 是解压缩到的目录,默认为当前目录;format 是压缩文件的格式,如果没有指定,则使用 file_name 的扩展名。例如:

```
>>> import shutil, os
>>> shutil.unpack_archive(r'c:\tmp\pybak.zip', extract_dir=r'c:\tmp')
>>> os.listdir(r'c:\tmp')
```

20.6 configparser 模块和配置文件

configparser 模块用于读取和写入配置文件。

20.6.1 INI 文件及 INI 文件格式

INI 文件即 Initialization File(初始化文件),也称为配置文件,其扩展名一般为.ini 或.cfg。INI 文件是文本格式的文件,通常位于应用程序的配置文件的文件夹中,用于保存应用程序的各种配置信息。

当应用程序启动时,会根据 INI 中的参数重新初始化应用程序的配置;在系统关闭之前,会将应用程序当前所需的全部配置保存到 INI 文件中。

INI 文件的内容由节(Section)、键(Option)和值(Value)组成。键和值对以=或:关联。注解以#号或;号开始,直到该行结尾均为注解。其基本格式为:

```
;注解行
[Section1 Name]
Option11 = Value11
Option12 = Value12
...
#注解内容
[Section2 Name]
Option21:Value21
Option22:Value22
...
```

例如,示例 config.ini 的内容为:

```
; config.ini file
[SystemInfo]
port = 8080
[GameInfo]
level = 1
scores = 0
```

20.6.2 ConfigParser 对象和 INI 文件操作

configparser 模块的 ConfigParser()函数用于读取和写入 INI 文件,语法形式如下:

```
configparser.ConfigParser(defaults = None, dict_type = collections.OrderedDict, allow_no_value = False, delimiters = ('=', ':'), comment_prefixes = ('#', ';'), inline_comment_prefixes = None, strict = True, empty_lines_in_values = True, default_section = configparser.DEFAULTSECT, interpolation = BasicInterpolation())
```

创建 ConfigParser 对象的参数众多,对象方法 defaults()返回其默认值。

ConfigParser 对象主要包括以下方法:

- get(section, option, *, raw=False, vars=None[, fallback]):返回指定键的值。
- getint(section, option, *, raw=False, vars=None[, fallback]):返回指定键的值(整型)。
- getfloat(section, option, *, raw=False, vars=None[, fallback]):返回指定键的值(浮点数)。
- getboolean(section, option, *, raw=False, vars=None[, fallback]):返回指定键

的值（布尔值）。

- sections()：返回所有 section 的列表，不包括 defualt section。
- items(section, raw=False, vars=None)：返回所有项目的列表。
- options(section)：返回所有键的列表。
- has_section(section)：判断是否存在 section。
- add_section(section)：添加 section，若已经存在，则将导致 DuplicateSectionError。
- remove_section(section)：删除 section。
- set(section, option, value)：设置键的值。若 section 不存在，则将导致 NoSectionError。
- remove_option(section, option)：删除键。
- read(filenames, encoding=None)：从指定文件名读取并解析 INI 配置。
- read_file(f, source=None)：从指定文件对象 f 读取并解析 INI 配置。
- read_string(string, source='<string>')：从指定字符串读取并解析 INI 配置。
- read_dict(dictionary, source='<dict>')：从指定字典读取并解析 INI 配置。
- write(fileobject, space_around_delimiters=True)：写入到文件对象。

【例 20.3】 读取和写入 INI 文件示例（configparser.py）。

```
import configparser
def ini_create():                                    #创建 INI 文件
    config = configparser.ConfigParser()
    config['SystemInfo'] = {'port':'8080'}
    config['GameInfo'] = {'level':1, 'scores':0}
    with open('example.ini', 'w') as configfile:
        config.write(configfile)
def ini_read_write():                                #读取和设置 INI 文件
    config = configparser.ConfigParser()
    config.read('example.ini')
    config['SystemInfo']['port'] = '8088'
    config.set('GameInfo', 'scores', '1000')
    for item in config.items('GameInfo'): print(item)
    with open('example.ini', 'w') as configfile:
        config.write(configfile)
if __name__ == '__main__':
    ini_create()                                     #创建 INI 文件
    ini_read_write()                                 #读取和设置 INI 文件
```

程序运行结果如下。

```
('level', '1')
('scores', '1000')
```

20.7 应用举例

20.7.1 病毒扫描

病毒扫描程序周期性地、系统地扫描计算机文件系统中的每个文件，并检查它们是否感染了病毒。

病毒扫描程序根据文件是否包含病毒特征码来判断文件是否感染了特定的病毒。病毒特征码是由计算机安全专家识别的特定病毒中出现的字节序列，在未感染的文件中不太可能出

现，因而可以作为病毒存在的证据。

病毒扫描程序包含一个病毒特征列表，并且会定期自动更新。

实现一个简单的病毒扫描程序的思维和流程如下：

（1）定义病毒特征码字典 signatures，使用字典储存病毒名称及其与病毒特征码的映射关系。

（2）定义病毒扫描递归函数 scan(path, sig)，递归访问目录 path 中的所有文件（包括子文件夹中的文件）。基本情况为如果 path 是文件，则检查其是否存在病毒特征码，并打印出结果信息；递归情况为如果 path 是目录，则递归调用 scan(path, sig)。

【例 20.4】 病毒扫描器示例程序（virus_scanner.py）。

```
import os
virus_sig = {'lovebug':'xy5020g2h1azzx33', 'Blaster':'fdp3014k1ks6hgbc'}
def scan(path, virus_sig):
    """扫描文件夹 path 及其子文件夹中的所有文件，判断是否包含特征码"""
    if os.path.isfile(path):                    #基本情况，打开文件，查找特征码
        infile = open(path)
        content = infile.read()
        infile.close()
        for virus in virus_sig:
            #检查文件内容是否包含特征码
            if content.find(virus_sig[virus]) >= 0:
                print('{0}, found virus {1}'.format(path, virus))
        return
    #递归情况：递归调用
    for item in os.listdir(path):
        fullpath = os.path.join(path, item)
        scan(fullpath, virus_sig)
if __name__ == '__main__':
    scan('test', virus_sig)
```

说明：

（1）本例使用了简单的非真实的病毒特征码来演示，实际病毒特征码比较复杂，一般存在于二进制文件中。

（2）使用 os 模块的 os.listdir(path) 函数可以返回目录 path 中的所有文件和子文件夹；os.path.isfile(path) 可以判断 path 是否为文件；os.path.join(path, item) 可以合并一个新的路径并返回。

20.7.2 文件目录树

实现一个遍历并输出目录结构的程序的思维和流程如下：

（1）定义递归函数 tree(path, level)，递归访问目录 path 中的所有文件和子文件夹。基本情况为如果 path 是文件，则输出文件名；递归情况为如果 path 是文件夹，则输出文件夹，并递归调用 tree(path, level)。

（2）使用 level 参数表示递归的层，即文件夹的层。输出文件或文件夹的缩进由 level 决定。

【例 20.5】 文件目录树示例程序（path_tree.py）。

```
import os
def tree(path, level):
    #如果路径不存在，则返回 None
    if not os.path.exists(path): return None
    #基本情况：如果是文件，则输出文件名
    if os.path.isfile(path):
```

```
        fileName = os.path.basename(path)
        print('\t' * level + '├── ' + fileName)
    elif os.path.isdir(path):                    #递归情况
        print('\t' * level + '├── ' + path)
        for item in os.listdir(path):
            tree(os.path.join(path, item), level + 1)
if __name__ == '__main__':
    tree(r"c:\pythonpa", 0)
```

程序运行(部分)结果如图 20-1 所示。

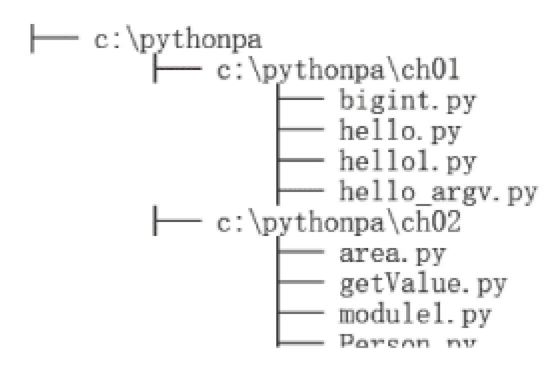


图 20-1 文件目录树示例程序的运行结果

20.8 复 习 题

一、填空题

1. 使用 Python 的 os 模块中的________函数可以创建目录。
2. 使用 Python 的________模块中的相关函数可以创建临时目录和文件。
3. 使用 Python 的 os 模块中的________函数可以切换当前工作目录。
4. 使用 Python 的 os 模块中的________函数可以显示一个目录中的文件/子目录列表。
5. 使用 Python 的 glob 模块中的________函数可以获取满足指定模式的文件/目录列表。
6. 使用 Python 的 os 模块中的________函数可以遍历指定的目录结构。
7. 使用 Python 的 os 模块中的________函数可以将目录名和文件名连接成全限定路径。
8. 使用 Python 的 os. path 模块中的________函数可以判断文件/目录是否存在。
9. 使用 Python 的 os. path 模块中的________函数可以判断路径 path 是否为文件类型。
10. 使用 Python 的 os. path 模块中的________函数可以判断路径 path 是否为目录类型。
11. 使用 Python 的 os. path 模块中的________函数可以判断路径 path 是否为绝对路径。
12. 使用 Python 的 os. path 模块中的________函数可以判断路径 path 是否为链接类型。
13. 使用 Python 的 os. path 模块中的________函数可以获取指定文件和目录的上次访问时间。
14. 使用 Python 的 os. path 模块中的________函数可以获取指定文件和目录的上次修改时间。
15. 使用 Python 的 os. path 模块中的________函数可以获取指定文件和目录的创建时间。
16. 使用 Python 的 os. path 模块中的________函数可以获取指定路径 path 的大小。
17. 使用 Python 的 os 模块中的________函数可以删除指定文件。
18. 使用 Python 的 os 模块中的________函数可以删除指定目录。
19. 使用 Python 的 shutil 模块中的________函数可以删除指定目录及目录下的所有内容。

20. 使用 Python 的 shutil 模块中的________函数可以复制目录树。

21. 使用 Python 的 shutil 模块中的________函数可以移动文件/目录。

22. 使用 Python 的 shutil 模块中的________函数可以复制文件/目录,并且除了复制文件内容外,还复制文件许可权限。

23. 使用 Python 的 shutil 模块中的________函数可以复制所有元数据,包括创建时间和修改时间。

24. 使用 Python 的 shutil 模块中的________函数可以获取磁盘空间的使用情况。

25. 使用 Python 的 os 模块中的________函数可以在 Python 程序中执行操作系统的命令和脚本,或运行其他程序。

26. 使用 Python 的 os 模块中的________函数可以在 Python 程序中执行操作系统的命令和脚本。

27. 通过 Python 的 os 或 shutil 模块的________函数可以获取终端的大小,以方便输出内容的格式化操作。

28. 使用 Python 的 shutil 模块的________和________等函数可以实现文件的压缩和解压缩功能。

29. Python 的 shutil 模块的________和________函数返回支持的压缩和解压缩格式。

30. INI 文件即 Initialization File(初始化文件),也称为配置文件,其扩展名一般为________或________。

31. INI 文件的内容由________、________和________组成。键和值对以________或________关联。注解以________或________开始,直到该行结尾均为注解。

32. Python 的 configparser 模块的________函数用于读取和写入 INI 文件。

二、思考题

1. Python 标准库中包括哪些系统管理相关模块?
2. Python 如何实现文件和目录的删除、复制、重命名功能?
3. Python 如何实现文件的压缩和解压缩功能?
4. Python 如何实现配置文件的读取和写入功能?

20.9　上机实践

完成本章中的例 20.1～例 20.5,熟悉 Python 语言系统管理程序设计。

20.10　案例研究:简易图形用户界面压缩软件

本章案例研究通过一个简易图形用户界面压缩软件的设计与实现,帮助读者深入了解使用 wxPython 和 Python 系统管理包开发、使用系统应用程序的思维和流程。

本章案例研究的解题思路和源代码等以电子版形式提供,具体请扫描如下二维码。



案例研究

参 考 文 献

[1] Python Software Foundation. Python v3.7 documentation. http://Python.org/.
[2] 江红,余青松. Python 程序设计与算法基础教程[M]. 北京：清华大学出版社,2017.
[3] 江红,余青松. Python 程序设计教程[M]. 北京：清华大学出版社,2014.
[4] 江红,余青松. C#程序设计教程[M]. 3 版. 北京：清华大学出版社,2018.
[5] 余青松,江红. C#程序设计实验指导与习题测试[M]. 3 版. 北京：清华大学出版社,2018.
[6] Doug Hellmann. The Python 3 Standard Library by Example[M]. Addison-Wesley Professional,2017.
[7] Magnus Lie Hetland. Beginning Python：From Novice to Professional[M]. 3rd ed. Apress，2017.
[8] Ljubomir Perkovic. Introduction to Computing Using Python[M]. 2nd ed. John Wiley & Sons，Inc.,2015.
[9] Robert Sedgewick，Kevin Wayne，Robert Dondero. Introduction to Programming in Python：An Interdisciplinary Approach[M]. Addison-Wesley Professional,2015.
[10] Think Python：How to Think Like a Computer Scientist[M]. 2nd ed. O'Reilly Media.